SpartanMC
Users Manual

Table of Contents

List of Figures

List of Tables

1. Instruction Set Architecture

The SpartanMC uses two register addresses per instruction. The first operand (RD register) is automatically used as the destination of the operation. This slightly reduces the effectiveness of the compiler, but it is a reasonable decision with respect to the very limited instruction bit width of 18 bit.

The code efficiency is improved with an additional condition code register which is used to store the result of compare instructions (used for branches).

1.1. Instruction Types

The instruction set is composed of fixed 18 bit instructions grouped in the four types:

- R-Type (register)
- I-Type (immidiate)
- M-Type (memory)
- J-Type (jump)

1.1.1. R-Type

R-Type instructions are used for operations which takes two register values and computes a result, which is stored back into operand one.

OPC	RD	RS1	Func

17　　　　13 12　　9 8　　5 4　　　0
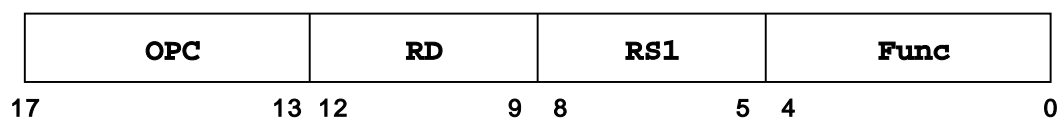
Figure 1-1: R-Type instruction

1.1.2. I-Type

This group includes all operations which take one register value and a constant to carry out an operation.

OPC	RD	Constant
17 13	12 9	8 0
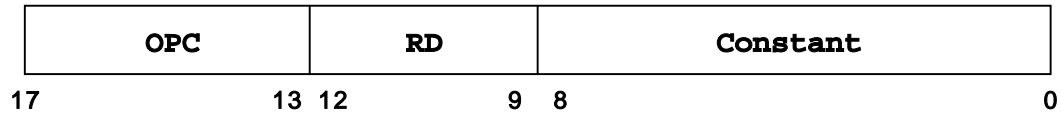
Figure 1-2: I-Type instruction

1.1.3. M-Type

This group is used for memory access operations. All load and store operations are available as half word (9 bit) or full word (18 bit) operation.

OPC	RD/RS2	RS1	Displ.
17 13	12 9	8 5	4 0
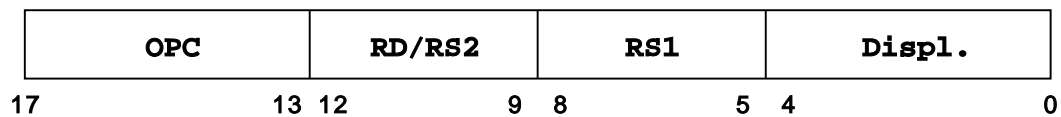
Figure 1-3: M-Type instruction

1.1.4. J-Type

This group includes the jump instruction and two branch instructions. The branch instructions interpret the condition code flag (see registers) to decide either to branch or not.

OPC	Offset
17 13	12 0
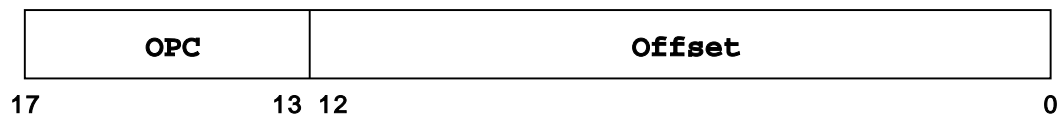
Figure 1-4: J-Type Instruction

1.2. Instruction Coding Matrices

The following table shows the instruction coding used on the SpartanMC.

Table 1-1: Main Matrix using IR 17-13

IR 17-13	..000	..001	..010	..011	..100	..101	..110	..111
00..	Special 1	Special 2	J	JALS	BEQZ	BNEZ	BEQZC	BNEZC
01..	ADDI	MOVI	LHI	SIGEX	ANDI	ORI	XORI	MULI
10..	L9	S9	L18	S18	SLLI	*	SRLI	SRAI
11..	SEQI	SNEI	SLTI	SGTI	SLEI	SGEI	IFADDUI	IFSUBUI

Table 1-2: Submatrix Special 1 using IR 4-0

IR 4-0	..000	..001	..010	..011	..100	..101	..110	..111
00..	orcc	andcc	*	*	SLL	MOV	SRL	SRA
01..	SEQU	SNEU	SLTU	SGTU	SLEU	SGEU	*	*
10..	*	*	*	*	*	*	CBITS	SBITS
11..	*	*	*	*	*	*	*	NOT

Table 1-3: Submatrix Special 2 using IR 4-0

IR 4-0	..000	..001	..010	..011	..100	..101	..110	..111
00..	RFE	TRAP	JR	JALR	JRS	JALRS	*	*
01..	*	*	*	*	*	*	*	*
10..	ADD	ADDU	SUB	SUBU	AND	OR	XOR	MUL
11..	SEQ	SNE	SLT	SGT	SLE	SGE	MOVI2S	MOVS2I

Note: * Code not used

Instructions written in lower case are currently not supported.

1.3. Register Window

The SpartanMC uses 16 addressable 18 bit registers which are stored in a 1k x 18 bit FPGA BlockRAM. The memory block is fully utilized through a sliding window technique. Registers 0 to 3 are used as global registers, registers 8 to 11 are local registers. The registers 4 to 7 are used as function input window for parameter transfer from the calling function. It equals registers 12 to 15 of the calling function which allows up to

four parameters for a function call without external memory. Each shift consumes eight positions in the block memory which results in a total of 127 call levels. Register 11 is reserved for the return address of subroutines or interupt service routines (ISR).
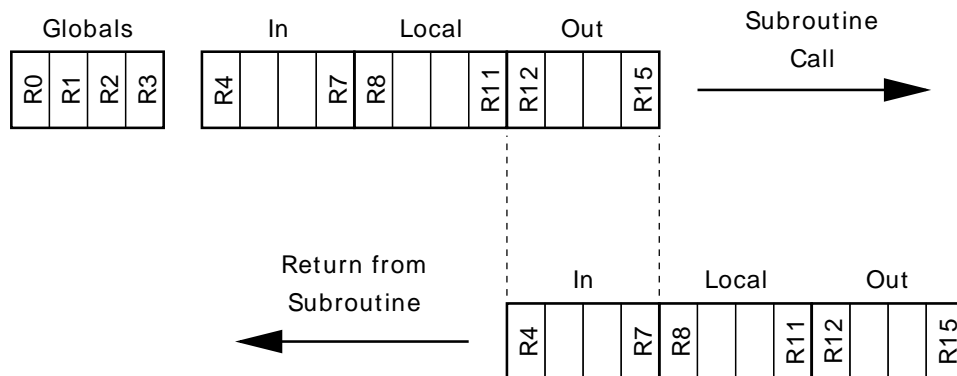
Figure 1-5: SpartanMC register window

1.4. Special Function Registers

For special purposes the SpartanMC contains special function registers (SFR). These registers could be modified via SBITS/CBITS instructions.

Note: The contents of all SFRs remain constant until the next access to the corresponding register value.

1.4.1. Status Register (SFR_STATUS)
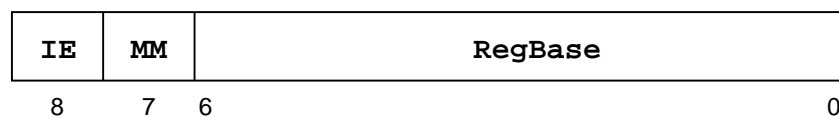
Figure 1-6: Status Register

SFR-Name: SFR_STATUS

SFR-Nr.: 0

SFR_STATUS [6:0]: Register Base (RegBase) - It contains the number of the current register window. The first window starts at 0. Each subroutine call increments the register by one up to the maximum value of 126.

SFR_STATUS [7]: Memory Management (MM) - This bit is set to 1 if the most significant address bit (address bit nr. 17) is used for memory access (see Address Management).

SFR_STATUS [8]: Interrupt Enable (IE) - If this bit is set to 0, the hardware interrupts are disabled. Setting IE to 1 enables the hardware interrupts.

1.4.2. LED Register (SFR_LEDS)

```
+-------------------------------------------------+
|                  LED output                     |
+-------------------------------------------------+
 6                                               0
```
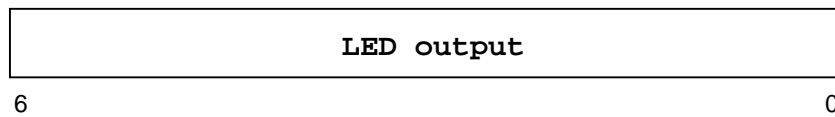
Figure 1-7: LED Register

SFR-Name: SFR_LEDS

SFR-Nr.: 1

SFR_LEDS [6:0]: This register is usable for custom status outputs.

1.4.3. MUL Register (SFR_MUL)

```
+-------------------------------------------------+
|        upper 18 bit multiplication result       |
+-------------------------------------------------+
 17                                              0
```
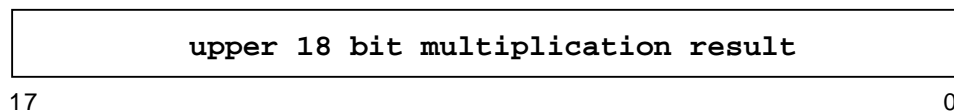
Figure 1-8: MUL Register

SFR-Name: SFR_MUL

SFR-Nr.: 2

SFR_MUL [17:0]: This register contains the upper 18 bit part [35:18] of a 36 bit result after a multiplication of two 18 bit values.

1.4.4. Condition Code Register (SFR_CC)

```
┌────┐
│ CC │
└────┘
  0
```
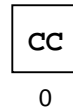
Figure 1-9: CC Register

SFR-Name: SFR_CC

SFR-Nr.: 3

SFR_CC [0]: Condition Code (CC) - The CC bit is used to store jump conditions. Furthermore it is used to signal an overflow after a signed arithmetic operation.

1.4.5. Interrupt Vector Register (SFR_IV)

```
┌─────────────────────────────────────────┐
│                    IV                    │
└─────────────────────────────────────────┘
17                                        0
```
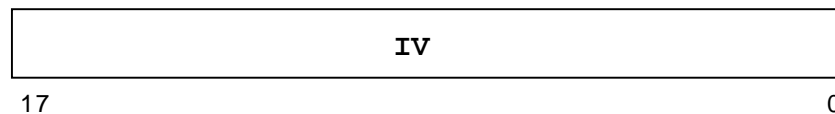
Figure 1-10: IV Register

SFR-Name: SFR_IV

SFR-Nr.: 4

SFR_IV [17:0]: Interrupt Vector (IV) - This Register contains the start address for the interrupt handling code (context switch and interrupt table lookup). After system reset this address is set to the value defined in the system configuration generated from jConfig. The start address of the interrupt handler can be changed by writing this register. This technique allows the usage of different interrupt service code for identical interrupts. It is recommended to disable the interrupts (set SFR_STATUS [8] to 0) befor writing SFR_IV.

1.4.6. Trap Vector Register (SFR_TR)

TR Base	TR Nr

17   8   7   0
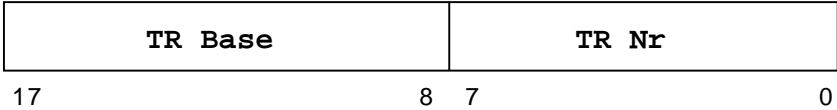
Figure 1-11: TR Register

SFR-Name: SFR_TR

SFR-Nr.: 5

This register contains the start address for trap service routines.

SFR_TR [17:8]: Trap (TR) - The upper 10 bits contain the base address of the trap table.

SFR_TR [7:0]: Trap (TR) - The lower 8 bits contain the number of the trap (read only - return 0x00 on read request). These bits are set via `trap` instruction.

1.4.7. Hardware Debugging Registers (SFR_DBG_IDX, SFR_DBG_DAT)

SFR_DBG_IDX
SFR_DBG_DAT

17   0
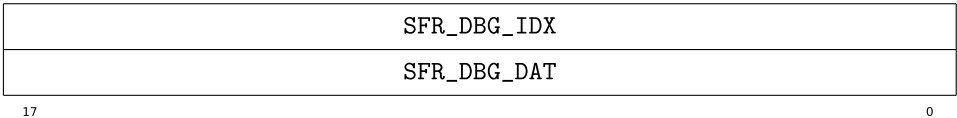
Figure 1-12: DBG Registers

SFR-Name: SFR_DBG_IDX, SFR_DBG_DAT

SFR-Nr.: 6,7

These registers allow indirect addressing of all Hardware Debugging registers

Both registers will be 0 if the Core was synthesized without hardware debugging support

See Hardware Debugging Support for more details on the indirect access

1.5. Instruction Set Details

This section is a reference to the entire SpartanMC instruction set.

Each of the following pages covers a single SpartanMC instruction. They are organized alphabetically by instruction mnemonic.

add   add

add

Mnemonic

add   Rd, Rs
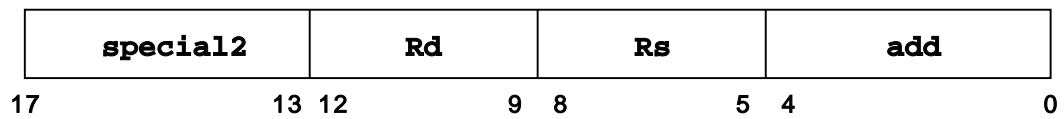
special2	Rd	Rs	add
17 13	12 9	8 5	4 0

Pseudocode

Rd ← Rd + Rs

CC ← OV

Description

The content of GPR Rd and the content of GPR Rs are arithmetically added and form an 18 bit two's complement result, which is written to GPR Rd. If the result of the addition is greater than 2^{17}-1 (i.e.: = 0x1FFFF) or lower than -2^{17}(i.e.: 0x20000), an overflow occurs and CC is set to 1.

Comments

R-Typ

addi   addi

add immediate

Mnemonic

addi   Rd, Imm

addi	Rd	Imm
17 13	12 9	8 0
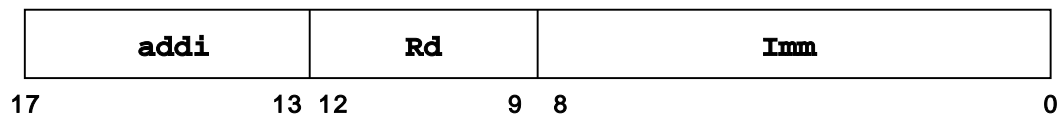
Pseudocode

$Rd \leftarrow Rd + IR_8^9 \#\# IR_{8:0}$

Description

The content of GPR Rd and the immediate Imm are arithmetically added and form an 18 bit two's complement result, which is written to GPR Rd.

Comments

I-Typ

$IR_8^9 \#\# IR_{8:0}$ performs a sign extension for the 9 bit immediate.

addu   addu

add unsigned

Mnemonic

addu   Rd, Rs
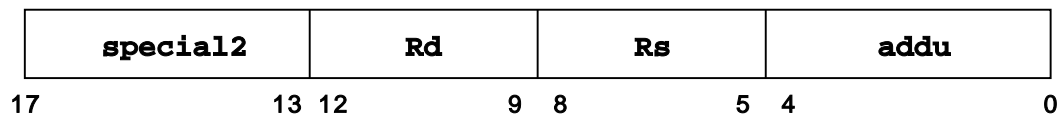
special2	Rd	Rs	addu

```
17            13 12      9 8      5 4        0
```

Pseudocode

Rd ← Rd + Rs

Description

The content of GPR Rd and the content of GPR Rs are arithmetically added and form
an 18-bit two's complement result which is written to GPR Rd.

Comments

R-Typ

and   and

and

Mnemonic

and   Rd, Rs

special2	Rd	Rs	and

17   13 12   9 8   5 4   0
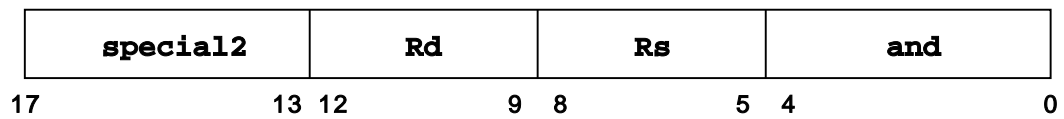
Pseudocode

Rd ← Rd and Rs

Description

The content of GPR Rd is combined with the content of GPR Rs in a bitwise logical AND operation. The result is written to GPR Rd.

Comments

R-Typ

andi   andi

and immediate

Mnemonic

andi   Rd, Imm

andi	Rd	Imm

17   13 12   9 8   0
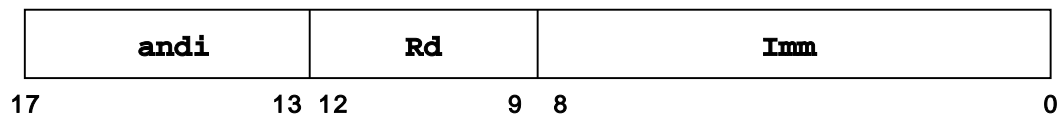
Pseudocode

$Rd \leftarrow Rd \text{ and } 0^9 \#\# IR_{8:0}$

Description

The zero extended 9 bit immediate is combined with the content of GPR Rd in a bitwise logical AND operation. The result is written to GPR Rd.

Comments

I-Typ

beqz

branch equal zero

Mnemonic

beqz   Rs, displacement
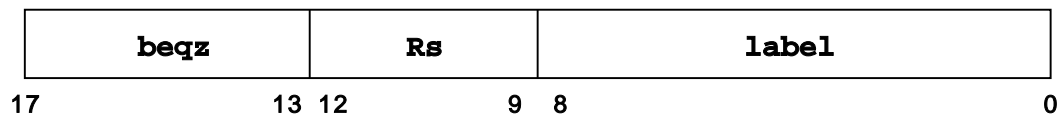
beqz	Rs	label
17 13	12 9	8 0

Pseudocode

IF Rs=0; PC ← PC + displacement

Delay Slots

1 unconditional delay slot

Description

Sets the program counter to PC + displacement, if GPR Rs equals zero. Note, that in contrast to all other relative jumps/branches the displacement has only a size of 9 bit instead of the usual 13 bit.

Comments

I-Typ

beqzc   beqzc

branch equal zero condition bit

Mnemonic

beqzc   displacement
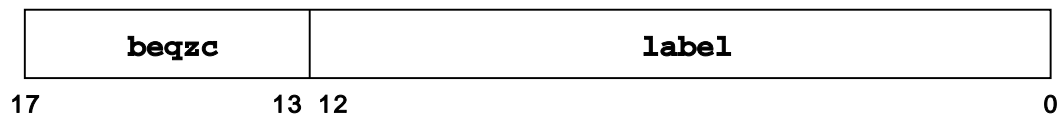
beqzc	label
17 13	12 0

Pseudocode

IF CC=0; PC ← PC + displacement

Delay Slots

1 unconditional delay slot

Description

Sets the program counter to PC + displacement if CC has a value of zero.

Comments

J-Typ

bnez   bnez

branch not equal zero

Mnemonic

bnez   Rs, displacement
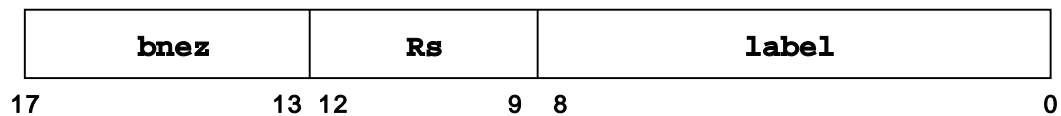
bnez	Rs	label
17 13	12 9	8 0

Pseudocode

IF Rs!=0; PC ← PC + displacement

Delay Slots

1 unconditional delay slot

Description

Sets the programm counter to PC + displacement, if GPR Rs is unequal to zero. Note, that in contrast to all other relative jumps/branches the displacement has only a size of 9 bit instead of the usual 13 bit.

Comments

I-Typ

bnezc

branch not equal zero condition bit

Mnemonic

bnezc   displacement
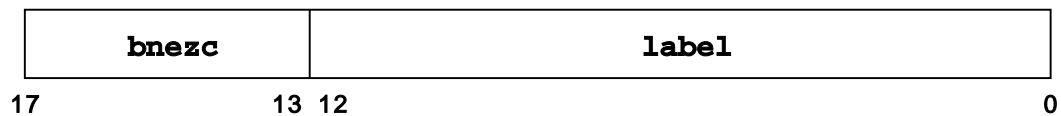
bnezc	label
17 13	12 0

Pseudocode

IF CC!=0; PC ← PC + displacement

Delay Slots

1 unconditional delay slot

Description

Sets the program counter to the PC + displacement if CC is unequal to zero.

Comments

J-Typ

cbits   cbits

clears bit at SFR

Mnemonic

cbits   BitNr

special1		BitNr	cbits
17 13	12 9	8 5	4 0
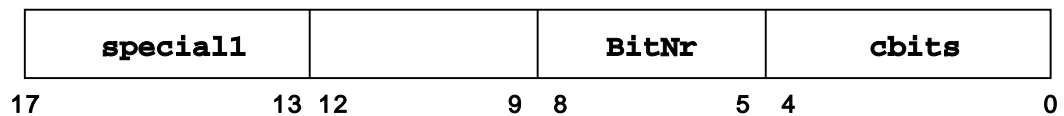
Pseudocode

```
SFR_Status_Bit ← 0
BitNr.: 0 = clears SFR_CC (CC)
BitNr.: 1 = clears SFR_STATUS_7(MM)
BitNr.: 2 = clears SFR_STATUS_8(IE)
```

Description

Clears a SFR bit according to the given BitNr. A BitNr of zero sets the CC bit to zero, a BitNr of one sets the MM bit to zero and a BitNr of two sets the IE bit to zero.

Comments

R-Typ

ifaddui

conditional addition with an unsigned immediate

ifaddui

Mnemonic

```
ifaddui     Rd, Imm
```

ifaddui	Rd	Imm

17   13 12   9 8   0
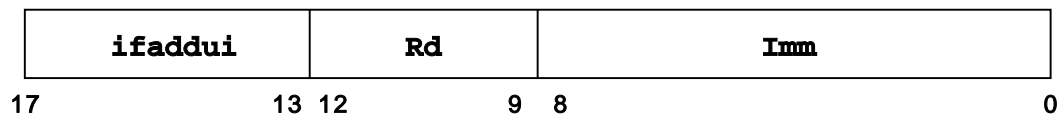
Pseudocode

IF CC = 1; Rd $\leftarrow$ Rd + 0^9 ## IR$_{8:0}$

Description

If the value of CC is one, the addition of the zero extended 9 bit immediate with the content of GPR Rd is carried out. The unsigned 18 bit result is written to GPR Rd. Otherwise GPR Rd remains unmodified.

Comments

I-Typ

ifsubui

ifsubui

conditional subtraction with an unsinged immediate

Mnemonic

```
ifsubui     Rd, Imm
```

ifsubui	Rd	Imm

17　　　　　　　13 12　　　9 8　　　　　　　　　　　　0
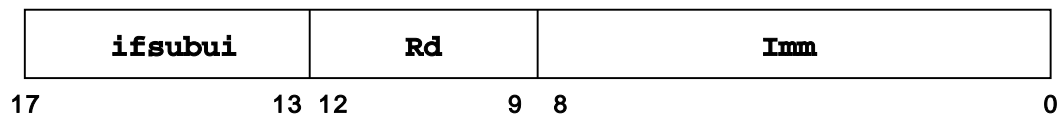
Pseudocode

$$\text{IF } CC=1; \; Rd \leftarrow Rd - 0^9 \, \#\# \, IR_{8:0}$$

Description

If the value of CC is one, the subtraction of the zero extended 9 bit immediate from the content of GPR Rd is carried out. The unsigned 18 bit result is written to GPR Rd.

Comments

I-Typ

j   j

jump

Mnemonic

```
j          displacement
```

j	label

17   13 12   0
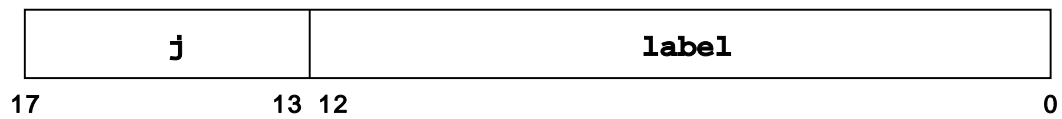
Pseudocode

```
PC ← PC + displacement
```

Delay Slots

1 unconditional delay slot

Description

Sets the PC unconditionally to the target address given with the value PC + displacement.

Comments

J-Typ

jalr

jump and link register

Mnemonic

```
jalr          Rs
```

special2		Rs	jalr
17 13	12 9	8 5	4 0
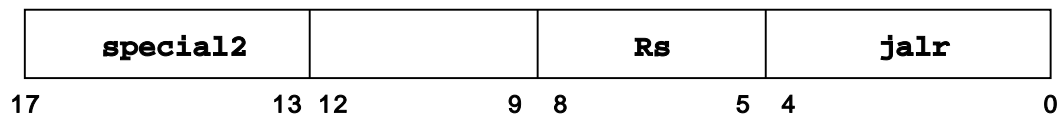
Pseudocode

```
R11 ← PC + 1
PC ← Rs
```

Delay Slots

1 unconditional delay slot

Description

Sets the program counter (PC) to the value of GPR Rs. The address of the instruction after the delay slot is written to GPR R11.

Comments

R-Typ

jalrs   jalrs

jump and link and shift register window

Mnemonic

```
jalrs        Rs
```

special2		Rs	jalrs
17 13	12 9	8 5	4 0
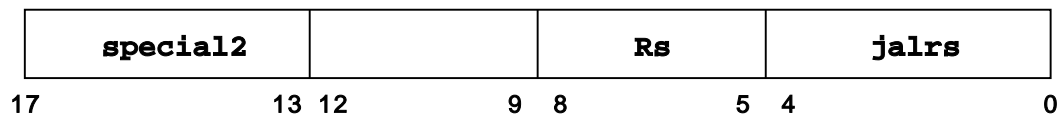
Pseudocode

```
RegBase ← RegBase + 1
R11 ← PC + 1
PC ← Rs
```

Delay Slots

1 unconditional delay slot

Description

This instruction performs a shift of the register window for eight register positions. This is used for subroutine calls. The current PC is incremented and stored in R11 of the new register window. R11 is used to store the return address of the calling function. The value for RegBase which holds the current subroutine call level (SFR_STATUS$_{6:0}$) is also incremented. Finally, the PC is set to the given address in GPR Rs.

Comments

R-Typ

jals

jump and link and shift register window

Mnemonic

```
jals        displacement
```

jals	label

17   13 12   0
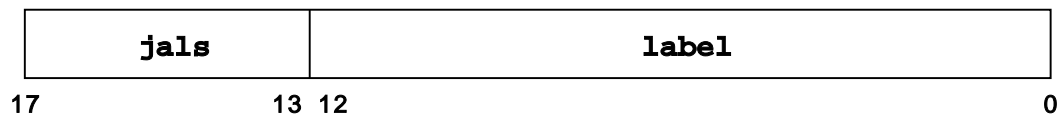
Pseudocode

```
RegBase ← RegBase + 1

R11 ← PC + 1

PC ← PC + displacement
```

Delay Slots

1 unconditional delay slot

Description

This instruction performs a shift of the register window for eight register positions. This is used for subroutine calls. The current PC is incremented and stored in R11 of the new register window. R11 is used to store the return address of the calling function. The value for RegBase which holds the current function call level ($SFR\_STATUS_{6:0}$) is also incremented. Finally, the PC is set to the PC + displacement.

Comments

J-Typ

The subroutine must have at least one instruction and the return code `jrs R11` at its end.

jr

jump register

Mnemonic

```
jr          Rs
```

special2		Rs	jr
17 13	12 9	8 5	4 0
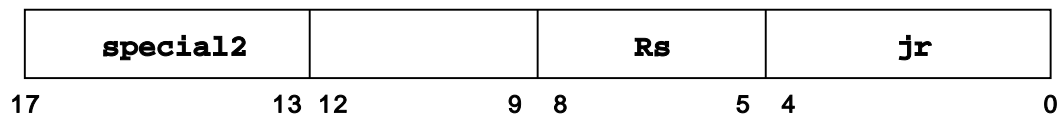
Pseudocode

PC ← Rs

Delay Slots

1 unconditional delay slot

Description

Set the PC unconditinally to the content of GPR Rs.

Comments

R-Typ

jrs

jump register shift register window (return subroutine)

jrs

Mnemonic

jrs   Rs
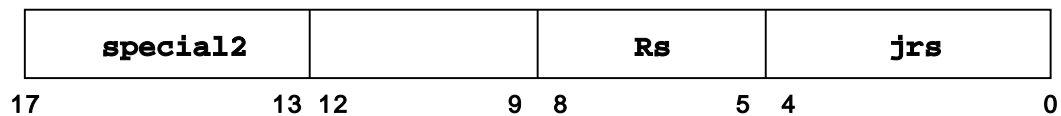
special2		Rs	jrs
17 13	12 9	8 5	4 0

Pseudocode

PC ← Rs

RegBase ← RegBase – 1

Delay Slots

1 unconditional delay slot

Description

This instruction performs the return from a subroutine by a back-shift of the register window for eight register positions. The program counter (PC) is set to the content of GPR RS.

Comments

R-Typ

l18

load 18 bit from memory

l18

Mnemonic

```
l18        Rd, disp(Rs)
```

l18	Rd	Rs	displ.
17 13	12 9	8 5	4 0
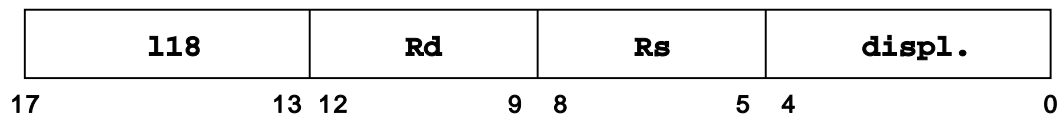
Pseudocode

Rd ← M[disp+Rs] ## M[disp+Rs+1]

Description

This instruction loads a sequence of two 9 bit words to an 18 bit register. The 5 bit displacement (disp) is zero-extended and added to the content of GPR Rs to form an unsigned 18 bit address. The 9 bit content of this address and the successor address is written to GPR Rd.

Comments

M-Typ

The given address must be even.

l9

load 9 bit from memory

Mnemonic

l9   Rd, disp(Rs)

l9	Rd	Rs	displ.

17   13 12   9 8   5 4   0
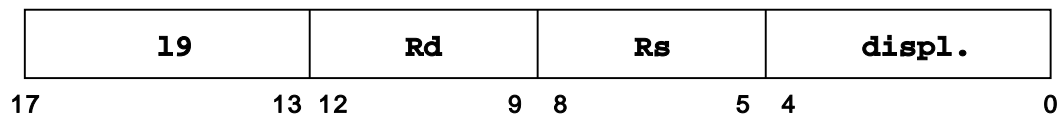
Pseudocode

Rd ← 0^9 ## M[disp+Rs]

Description

The 5 bit displacement (disp) is zero-extended and added to the content of GPR Rs to form an unsigned 18 bit address. The 9 Bit content of this address is written to GPR Rd.

Comments

M-Typ

The given address can be even or odd.

lhi

lhi

load high immediate

Mnemonic

lhi   Rd, Imm

lhi	Rd	Imm
17 13	12 9	8 0
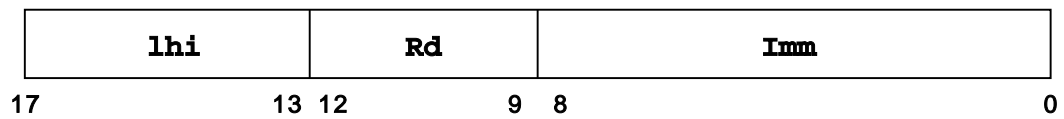
Pseudocode

$Rd \leftarrow IR_{8:0} \ \#\# \ 0^9$

Description

This instruction writes the upper 9 bit part of GPR Rd. Therefore, the 9 bit immediate is concatenated with a 9 bit zero value and written to GPR Rd.

Comments

I-Typ

mov

<div style="text-align: right;">

mov
</div>

move

Mnemonic

mov   Rd, Rs

special1	Rd	Rs	mov

17   13 12   9 8   5 4   0
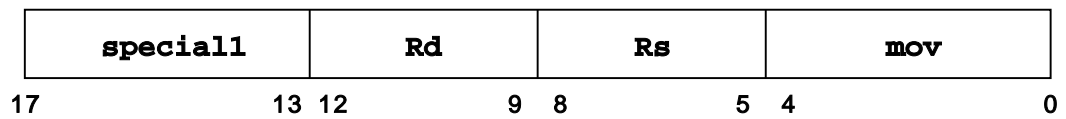
Pseudocode

Rd ← Rs

Description

The content of GPR Rs is written to GPR Rd.

Comments

R-Typ

movi   movi

move immediate

Mnemonic

movi   Rd, Imm

movi	Rd	Imm
17 13	12 9	8 0
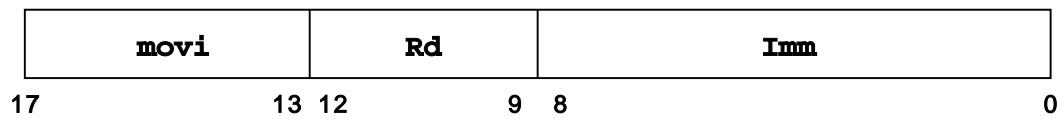
Pseudocode

$Rd \leftarrow 0^9 \ \#\# \ IR_{8:0}$

Description

The content of a zero-extended 9 bit immediate is written to GPR Rd.

Comments

I-Typ

movi2s movi2s

move integer to special

Mnemonic

movi2s   SfrNr, Rs

special2	Rs	SfrNr	movi2s
17 13	12 9	8 5	4 0
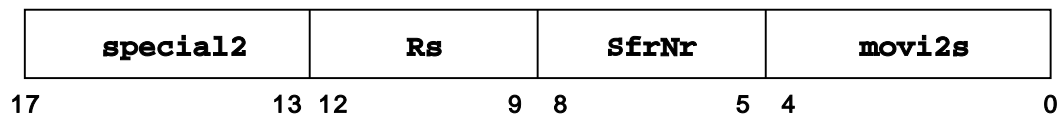
Pseudocode

SFR ← Rs

Description

The content of GPR Rs is written to the SFR with the given SfrNr.

If the destination SFR is SFR_Status, a register window change will only take effect after the next instruction. This is the same behaviour as with delay slots in Function Calls, where the delay slot still uses the old register window.

Comments

R-Typ

movs2i

movs2i

move from special register to integer

Mnemonic

movs2i   Rd, SfrNr

special2	Rd	SfrNr	movs2i
17 13	12 9	8 5	4 0
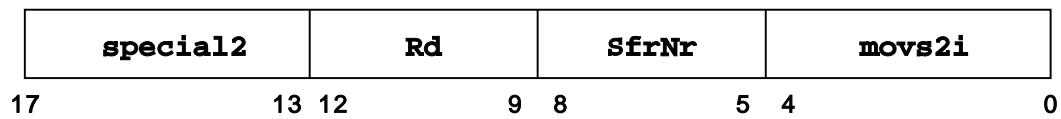
Pseudocode

Rd ← SFR

Description

The content of the SFR with SfrNr is written to GPR Rd.

Comments

R-Typ

In this instruction code, $IR_{8:5}$ holds the number of the SFR which is used as destination register for this instruction. The source register is given in $IR_{12:9}$.

mul

mul

multiply

Mnemonic

```
mul          Rd, Rs
```

special2	Rd	Rs	mul
17 13	12 9	8 5	4 0
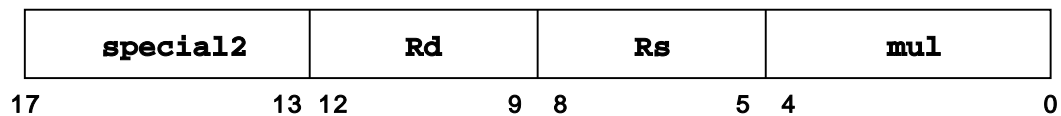
Pseudocode

SFR_MUL ## Rd ← Rd * Rs

Description

The content of GPR Rd and the content of GPR Rs are arithmetically multiplied, treating both operands as 18 bit two's complements values, and form a 36 bit two's complements result. The upper 18 bit part is written to SFR_MUL, the lower 18 bit part is written to GPR Rd.

Comments

R-Typ

muli   muli

multiply immediate

Mnemonic

```
muli        Rd, Imm
```

muli	Rd	Imm
17 13	12 9	8 0
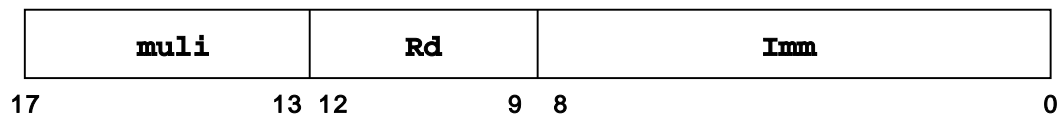
Pseudocode

$\text{SFR\_MUL} \,\#\#\, \text{Rd} \leftarrow \text{Rd} \times \text{IR}_8{}^9 \,\#\#\, \text{IR}_{8:0}$

Description

The sign-extended 9 bit immediate and the content of GPR Rd are arithmetically multiplied, treating both operants as 18 bit two's complement values, and form a 36-bit two's complement result. The upper 18 bit part is written to SFR_MUL, the lower 18 bit part is written to GPR Rd.

Comments

I-Typ

nop
no operation

Mnemonic

nop   Rd

special2	Rd	Rs	or

17   13 12   9 8   5 4   0
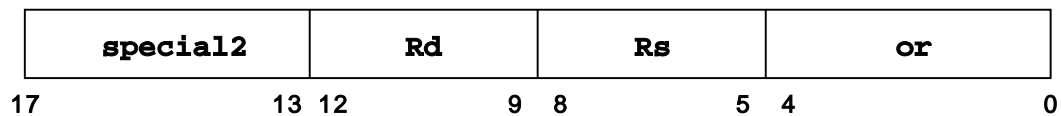
Pseudocode

Rd ← Rd or Rd

Description

Convenience Instruction. Is really an `or Rd, Rd` , but much more recognizable. Since there are 16 possible no-op combinations, this allows encoding extra information into the nop. This is useful only for debugging purposes, as it allows manipulating bypass logic and makes the instruction more distinguishable in assembler. Using no parameter defaults to 0.

Comments

R-Typ

not   not

not

Mnemonic

not   Rd, Rs

special1	Rd	Rs	not
17 13	12 9	8 5	4 0
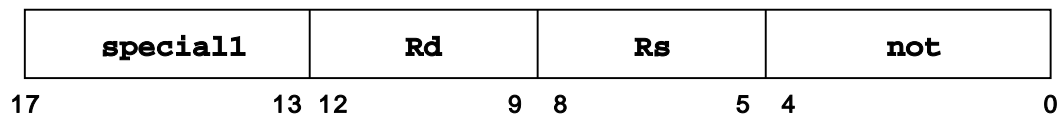
Pseudocode

Rd ← !Rs

Description

The content of GPR Rs is negated bitwise and the results is written to GPR Rd.

Comments

R-Typ

or   or

or

Mnemonic

`or          Rd, Rs`

special2	Rd	Rs	or
17 13	12 9	8 5	4 0
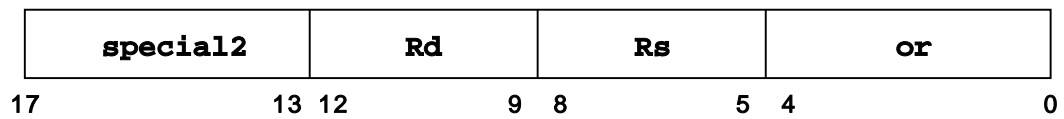
Pseudocode

`Rd ← Rd or Rs`

Description

The content of GPR Rs is combined with the content of GPR Rd in a bitwise logical OR operation, and the result is written to GPR Rd.

Comments

R-Typ

ori

ori

or immediate

Mnemonic

```
ori         Rd, Imm
```

ori	Rd	Imm

17　　　　　　13 12　　　9 8　　　　　　　　　　　　　　0

Pseudocode

$$Rd \leftarrow Rd \ or \ 0^9 \ \#\# \ IR_{8:0}$$

Description

The zero-extended 9 bit immediate is combined with the content of GPR Rd in a bitwise OR operation, and the result is written to GPR Rd.

Comments

I-Typ

rfe

return from exception

Mnemonic

rfe   Rs
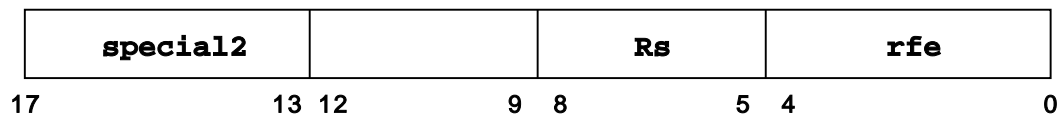
special2		Rs	rfe

17   13 12   9 8   5 4   0

Pseudocode

PC ← Rs

RegBase ← RegBase - 1

Delay Slots

1 unconditional delay slot

Description

This instruction performs the return from interrupt handling by a back-shift of the register window for eight register positions. The program counter (PC) is set to the content of GPR Rd and the interrupt is acknowledged.

This instruction is internally identical to `jsr` except that this will raise `ir_return` for a single cycle.

Comments

R-Typ

s18

store 18 bit to memory

s18

Mnemonic

s18   disp(Rs2), Rs1

s18	Rs1	Rs2	displ.

17   13 12   9 8   5 4   0
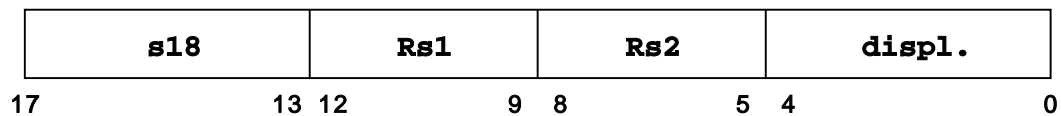
Pseudocode

$\text{M[disp+Rs2]} \mathbin{\#\#} \text{M[disp+Rs2+1]} \leftarrow \text{Rs1}$

Description

The zero-extended 5 bit displacement (disp) is added to the content of GPR Rs2 to form an unsigned 18 bit address. The content of GPR Rs1 is stored at this address.

Comments

M-Typ

The 18 bit address must be even.

s9

store 9 Bit to memory

Mnemonic

s9 disp(Rs2), Rs1

s9	**Rs1**	**Rs2**	**displ.**

17   13 12   9 8   5 4   0
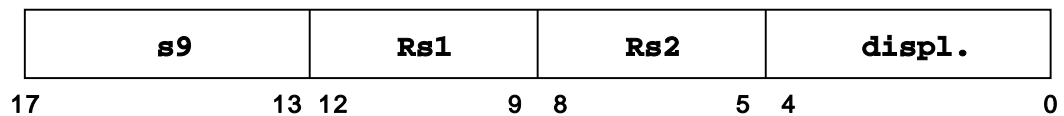
Pseudocode

M[disp+Rs2] ← Rs1

Description

The 5 bit displacement (disp) is zero-extended and added to the content of GPR Rs2 to form an unsigned 18 bit address. The lower 9 bit part of GPR Rs1 is stored at this address.

Comments

M-Typ

The given address can be even or odd.

sbits sbits

set bit at SFR-Register

Mnemonic

sbits   BitNr

special1		BitNr	sbits
17 13	12 9	8 5	4 0
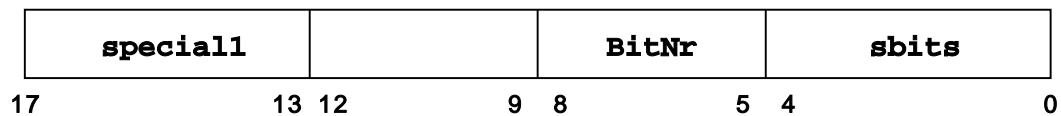
Pseudocode

```
SFR_Status_Bit ← 1
BitNr.: 0 = sets SFR_CC
BitNr.: 1 = sets SFR_STATUS₇(MM)
BitNr.: 2 = sets SFR_STATUS₈(IE)
```

Description

This instruction sets a SFR bit according to the given BitNr. A BitNr of zero sets the CC bit to one, a BitNr of one sets the MM bit to one and a BitNr of two sets the IE bit to one.

Comments

R-Typ

seq

set equal

seq

Mnemonic

seq   Rs2, Rs1
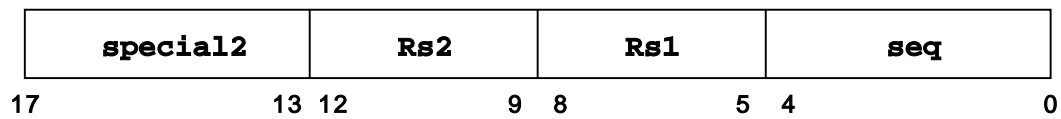
special2	Rs2	Rs1	seq

17   13 12   9 8   5 4   0

Pseudocode

CC $\leftarrow$ Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If both values are equal, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{17}-1 and greater than or equal to -2^{17}.

Comments

R-Typ

seqi   seqi

set equal immediate

Mnemonic

seqi   Rs, Imm
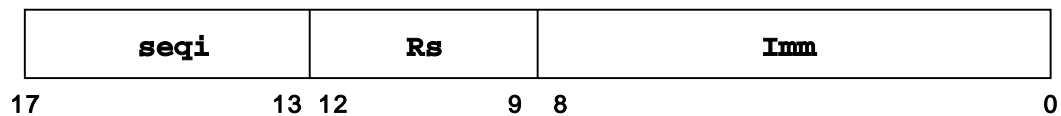
seqi	Rs	Imm

17   13 12   9 8   0

Pseudocode

$$CC \leftarrow Rs - IR_8{}^9 \ \#\# \ IR_{8:0}$$

Description

This instruction compares the content of GPR Rs and the 9 bit immediate $IR_8{}^9 \#\# \ IR_{8:0}$. If both values are equal, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The content of GPR Rs is lower than or equal to $2^{17}-1$ and greater than or equal to -2^{17}.

Comments

I-Typ

sequ

set equal unsigned

Mnemonic

sequ   Rs2, Rs1
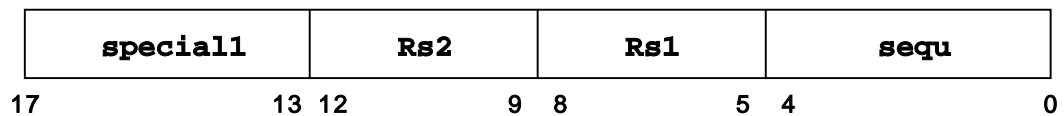
special1	Rs2	Rs1	sequ
17 13	12 9	8 5	4 0

Pseudocode

CC ← Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If both values are equal, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{17}-1 and greater than or equal to zero.

Comments

R-Typ

sge

set greater than or equal

<div style="text-align: right">

sge

</div>

Mnemonic

sge   Rs2, Rs1

special2	Rs2	Rs1	sge
17 13	12 9	8 5	4 0

Pseudocode

CC ← Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is equal to or greater than the value of Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{17}-1 and greater than or equal to -2^{17}.

Comments

R-Typ

sgei   sgei

set greater than or equal immediate

Mnemonic

sgei   Rs, Imm

sgei	Rs	Imm
17 13	12 9	8 0

Pseudocode

$$CC \leftarrow Rs - IR_8^9 \text{ \#\# } IR_{8:0}$$

Description

This instruction compares the content of GPR Rs and the content of a 9 bit immediate. If the value of Rs is equal to or greater than the immediate, the result will be one, otherwise the reslt will be zero. The 18-bit result is written to SFR_CC. The content of GPR Rs is lower than or equal to $2^{17}-1$ and greater than or equal to -2^{17}.

Comments

I-Typ

sgeu   sgeu

set greater than or equal unsigned

Mnemonic

sgeu   Rs2, Rs1
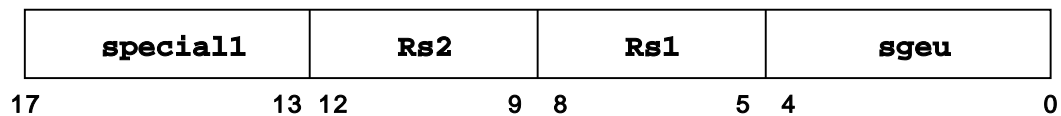
special1	Rs2	Rs1	sgeu

17   13 12   9 8   5 4   0

Pseudocode

CC ← Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is equal to or greater than the value of Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{18}-1 and greater than or equal to zero.

Comments

R-Typ

In this instruction -1 = 0x3FFFF is bigger than Ox1FFFF.

sgt

set greater than

Mnemonic

sgt   Rs2, Rs1
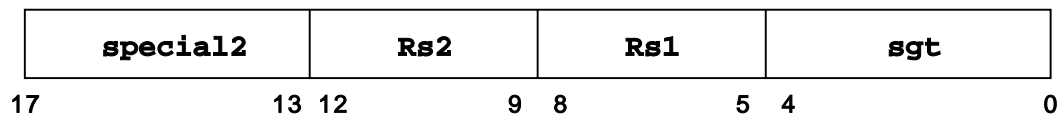
special2	Rs2	Rs1	sgt

17   13 12   9 8   5 4   0

Pseudocode

CC ← Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of GPR Rs2 is greater than the value of GPR Rs1, the result will be one, otherwise, the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than 2^{17}-1 and greater than -2^{17}.

Comments

R-Typ

sgti

sgti

set greater than immediate

Mnemonic

sgti   Rs, Imm
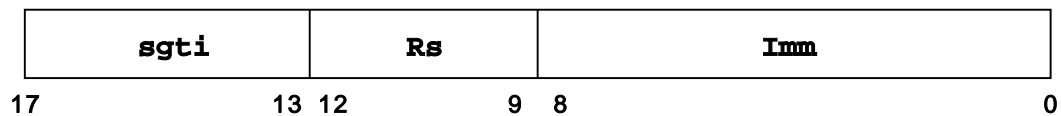
sgti	Rs	Imm
17 13	12 9	8 0

Pseudocode

$$CC \leftarrow Rs - IR_8{}^9 \ \#\# \ IR_{8:0}$$

Description

This instruction compares the content of GPR Rs and a 9 bit immediate. If the value of GPR Rs is greater than the immediate, the result will be one, otherwise the result will be zero. This result is written to SFR_CC. The content of GPR Rs is lower than or equal to $2^{17}-1$ and greater than or equal to -2^{17}.

Comments

I-Typ

sgtu

set greather than unsigned

Mnemonic

sgtu   Rs2, Rs1
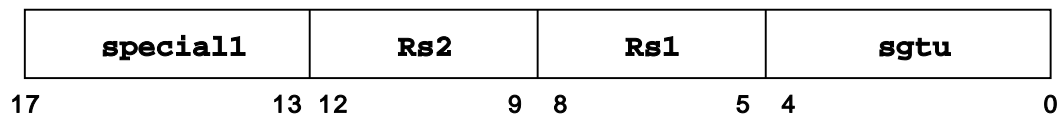
special1	Rs2	Rs1	sgtu

17   13 12   9 8   5 4   0

Pseudocode

CC ← Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is greater than the value of Rs1, the result will be one, otherwise, the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to $2^{18}-1$ and greater than or equal to zero.

Comments

R-Typ

sigex

sigex

signum extention

Mnemonic

sigex   Rd, BitNr

sigex	Rd	BitNr
17 13	12 9	8 0
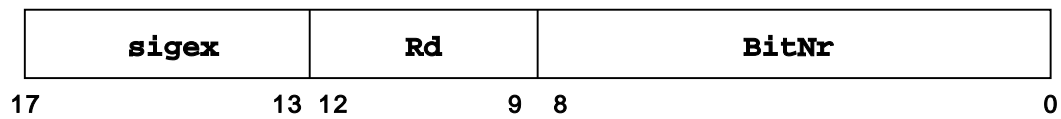
Pseudocode

$$Rd \leftarrow Rd_{BitNr-1}^{17-BitNr-1} \,\#\#\, Rd_{(BitNr-1):0}$$

Description

This instruction expands the content of Rd to an 18 bit value using the value of Rd at the given bit number (BitNr).

Comments

I-Typ

The allowed values for BitNr are 8, 9 or 16. Other values will be treated as 8.

sle

<div align="right">

sle

</div>

set less than or equal

Mnemonic

sle   Rs2, Rs1
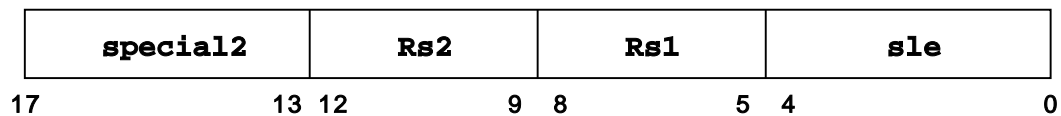
special2	Rs2	Rs1	sle
17 13	12 9	8 5	4 0

Pseudocode

CC ← Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is equal to or less than the value of Rs2, the result will be one, otherwise, the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{17}-1 and greater than or equal to -2^{17}.

Comments

R-Typ

slei slei

set less than or equal immediate

Mnemonic

```
slei        Rs, Imm
```
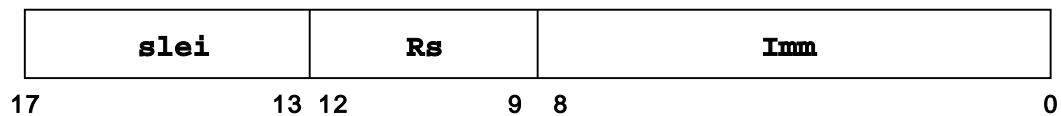
slei	Rs	Imm
17 13	12 9	8 0

Pseudocode

$$CC \leftarrow Rs - IR_8{}^9 \,\#\#\, IR_{8:0}$$

Description

This instruction compares the content of GPR Rs and a 9 bit immediate. If the value of Rs is equal to or less than the immediate, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The content of GPR Rs is lower than or equal to $2^{17}-1$ and greater than or equal to -2^{17}.

Comments

I-Typ

sleu sleu

set less than or equal unsigned

Mnemonic

sleu   Rs2, Rs1
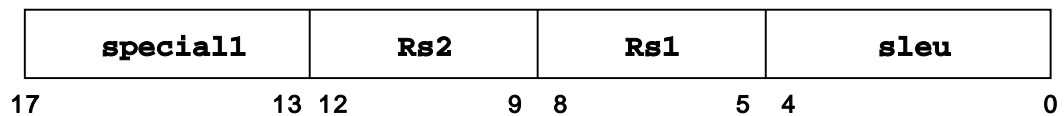
special1	Rs2	Rs1	sleu

17   13 12   9 8   5 4   0

Pseudocode

CC $\leftarrow$ Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is equal to or less than the value of Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{18}-1 and greater than or equal to zero.

Comments

R-Typ

slt

set less than

<div style="text-align: right">

slt

</div>

Mnemonic

```
slt        Rs2, Rs1
```
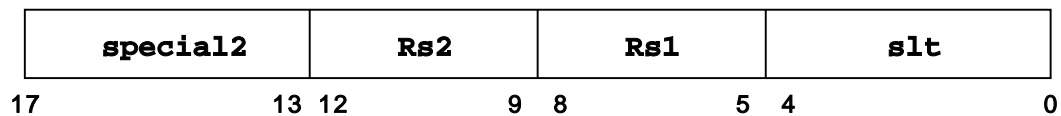
special2	Rs2	Rs1	slt
17 13	12 9	8 5	4 0

Pseudocode

```
CC ← Rs2 - Rs1
```

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of GPR Rs2 is less than the value of GPR Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than 2^{17}-1 and greater than -2^{17}.

Comments

R-Typ

slti

slti

set less than immediate

Mnemonic

slti   Rs, Imm
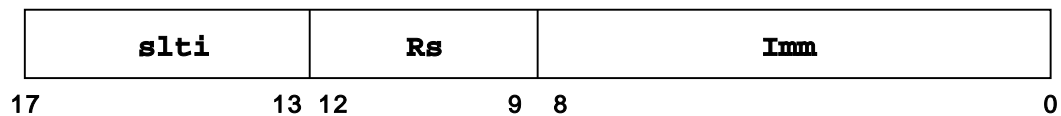
slti	Rs	Imm
17 13	12 9	8 0

Pseudocode

$CC \leftarrow Rs - IR_8{}^9 \#\# IR_{8:0}$

Description

This instruction compares the content of GPR Rs and a 9 bit immediate. If the value of Rs is less than the immediate, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The content of GPR Rs is lower than or equal to $2^{17}-1$ and greater than or equal to -2^{17}.

Comments

I-Typ

sltu   sltu

set less than unsigned

Mnemonic

sltu   Rs2, Rs1
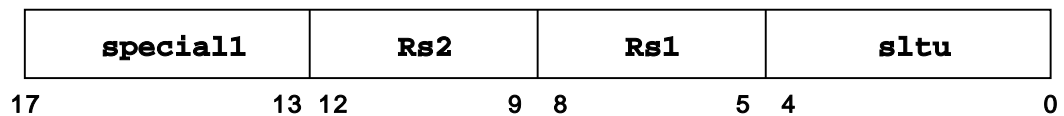
special1	Rs2	Rs1	sltu

17   13 12   9 8   5 4   0

Pseudocode

CC ← Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is less than the value of Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{18}-1 and greater than or equal to zero.

Comments

R-Typ

sne sne

set not equal

Mnemonic

sne   Rs2, Rs1
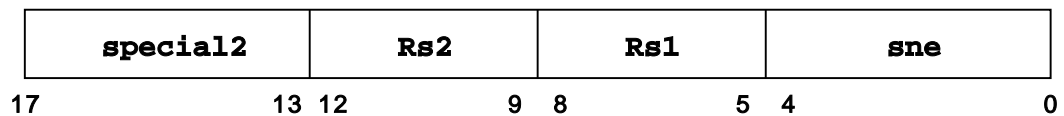
special2	Rs2	Rs1	sne
17 13	12 9	8 5	4 0

Pseudocode

CC ← Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of GPR Rs2 is lower or greater than the value of GPR Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than 2^{17}-1 and greater than -2^{17}.

Comments

R-Typ

snei

snei

set not equal immediate

Mnemonic

snei   Rs, Imm
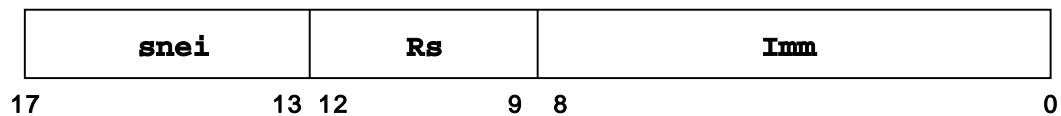
snei	Rs	Imm

17   13 12   9 8   0

Pseudocode

$CC \leftarrow Rs - IR_8{}^9 \#\# IR_{8:0}$

Description

This instruction compares the content of GPR Rs and the content of 9 bit immediate. If the value of Rs is greater or lower than the immediate, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The content of GPR Rs is lower than or equal to $2^{17}-1$ and greater than or equal to -2^{17}.

Comments

I-Typ

sneu   sneu

set not equal unsigned

Mnemonic

sneu   Rs2, Rs1
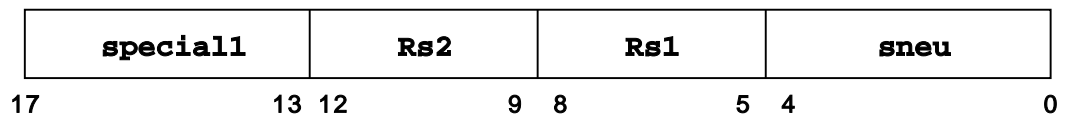
special1	Rs2	Rs1	sneu

17   13 12   9 8   5 4   0

Pseudocode

CC ← Rs2 - Rs1

Description

This instruction compares the content of GPR Rs2 and GPR Rs1. If the value of Rs2 is lower or greater than the value of Rs1, the result will be one, otherwise the result will be zero. The result is written to SFR_CC. The contents of GPR Rs2 and GPR Rs1 are lower than or equal to 2^{18}-1 and greater than or equal to zero.

Comments

R-Typ

sub   sub
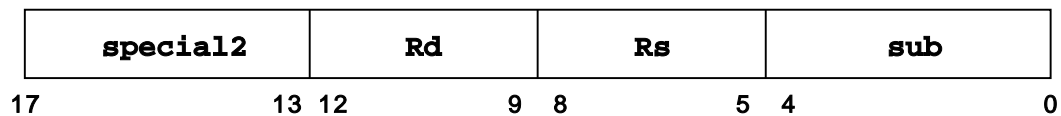
subtract

Mnemonic

```
sub          Rd, Rs
```

special2	Rd	Rs	sub
17 13	12 9	8 5	4 0

Pseudocode

```
Rd ← Rd - Rs
cc ← ov
```

Description

The content of GPR Rs is arithmetically subtracted from the content of GPR Rd and forms an 18 bit two's complement result, which is written to GPR Rd. If the result of the subtraction is greater than 2^{17}-1 (i.e.: = 0x1FFFF) or lower than -2^{17}(i.e.: 0x20000), an overflow occurs and CC is set to 1.

Comments

R-Typ

subu

subtract unsigned

Mnemonic

`subu`   Rd, Rs
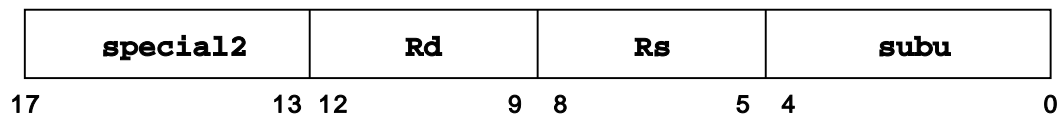
special2	Rd	Rs	subu
17 13	12 9	8 5	4 0

Pseudocode

Rd ← Rd - Rs

Description

The content of GPR Rs is arithmetically subtracted from the content of GPR Rd and forms an 18 bit unsigned result which is written to GPR Rd. This instruction can not produce an overflow exception, which is the only difference between this instruction and the sub instruction.

Comments

R-Typ

sll

shift left logical

Mnemonic

sll   Rd, Rs

special1	Rd	Rs	sll

17   13 12   9 8   5 4   0
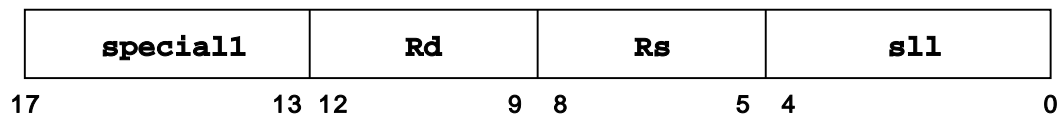
Pseudocode

Rd ← CC ## Rd « Rs

Description

This instruction performs a left shift operation of GPR Rd. The shift width is set to the value of GPR Rs. The free bit positions are filled with zeros. The value of the highest bit in GPR Rd is written to SFR_CC. The result is written to GPR Rd.

Comments

R-Typ

The value in GPR Rs will be ignored and the shift width will be always one if single shift is configured.

Figure 1-13: shift left logical

slli   slli

shift left logical immediate

Mnemonic

slli   Rd, Imm

slli	Rd	Imm
17 13	12 9	8 0
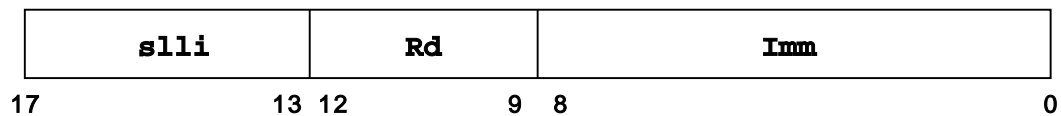
Pseudocode

$Rd \leftarrow (CC \,\#\# \,Rd) \ll (0^9 \,\#\# \,IR_{8:0})$

Description

This instruction performs a left shift operation of GPR Rd. The shift width is set by a zero-extended 9 bit immediate. The free bit positions are filled with zero. The value of the highest bit in GPR Rd is written to SFR_CC. The result is written to GPR Rd.

Comments

R-Typ

The value in GPR Rs will be ignored and the shift width will be always one if single shift is configured.

CC	←	GPR	←	0
		17 0		
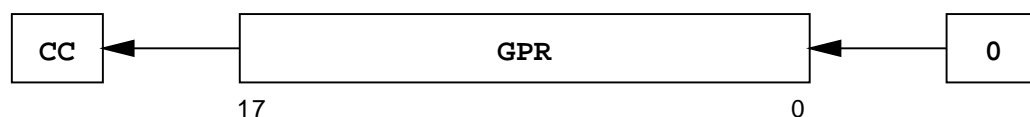
Figure 1-14: shift left logical immediate

srl

shift right logical

<div align="right">

srl
</div>

Mnemonic

srl   Rd, Rs

special1	Rd	Rs	srl
17 13	12 9	8 5	4 0
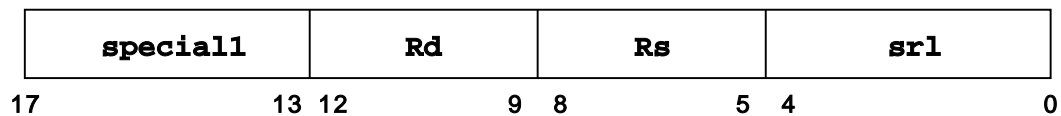
Pseudocode

Rd ← (Rd ## CC) » Rs

Description

This instruction performs a right shift operation of GPR Rd. The shift width is set to the value of GPR Rs. The free bit positions are filled with zeros. The value of the lowest bit in GPR Rd is written to SFR_CC. The result is written to GPR Rd.

Comments

R-Typ

The value in GPR Rs will be ignored and the shift width will be always one if single shift is configured.

0	→	GPR	→	CC
		17 0		
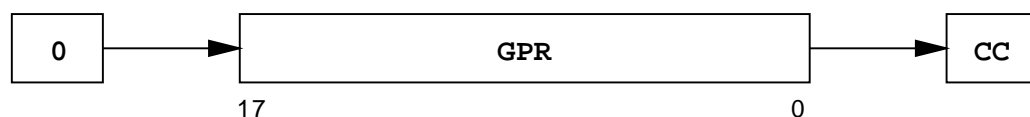
Figure 1-15: shift right logical

srli

shift right logical immediate

<div style="text-align: right">

srli
</div>

Mnemonic

srli   Rd, Imm

srli	Rd	Imm
17 13	12 9	8 0
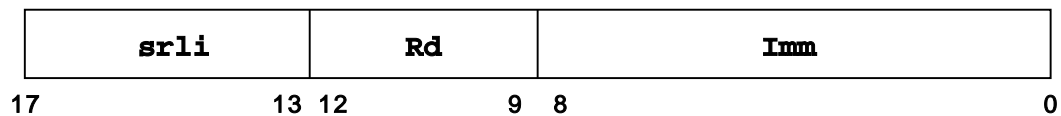
Pseudocode

$Rd \leftarrow Rd ## CC \gg 0^9 ## IR_{8:0}$

Description

This instruction performs a right shift operation of GPR Rd. The shift width is set by a zero-extended 9 bit immediate. The free bit positions are filled with zero. The value of the lowest bit in GPR Rd is written to SFR_CC. The result is written to GPR Rd.

Comments

R-Typ

The value in GPR Rs will be ignored and the shift width will be always one if single shift is configured.

0	GPR	CC
	17 0	
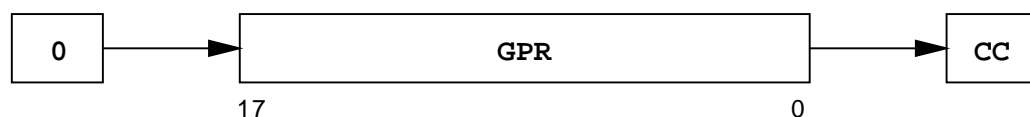
Figure 1-16: shift right logical immediate

sra

<div align="right">

sra

</div>

shift right arithmetic

Mnemonic

sra   Rd, Rs

special1	Rd	Rs	sra
17 13	12 9	8 5	4 0
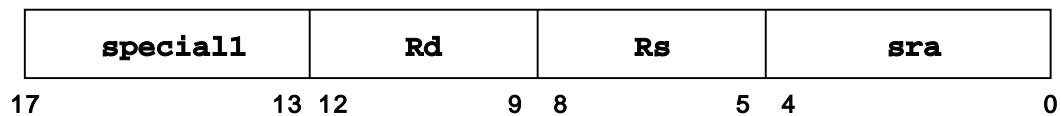
Pseudocode

$Rd \leftarrow (Rd \#\# CC) \gg_a Rs$

Description

This instruction performs a right shift operation of GPR Rd. The shift width is set to the value of GPR Rs. The free bit positions are filled with the highest bit of GPR Rd. The value of the lowest bit in GPR Rd is written to SFR_CC. The result is written to GPR Rd.

Comments

R-Typ

The value in GPR Rs will be ignored and the shift width will be always one if single shift is configured.
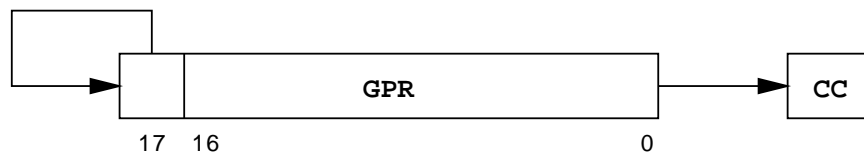
Figure 1-17: shift right arithmetic

srai   srai

shift right arithmetic immediate

Mnemonic

srai   Rd, Imm

srai	Rd	Imm
17 13	12 9	8 0
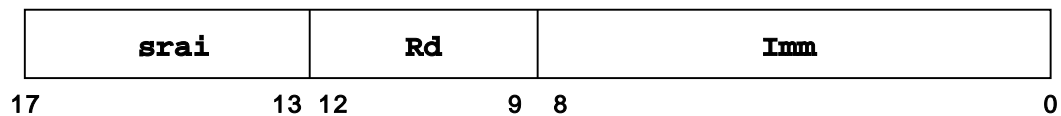
Pseudocode

$Rd \leftarrow Rd \ \#\# \ CC \ \gg_a 0^9 \ \#\# \ IR_{8:0}$

Description

This instruction performs a right shift operation of GPR Rd. The shift width is set by a zero-extended 9 bit immediate. The free bit positions are filled with the highest bit of GPR Rd. The value of the lowest bit in GPR Rd is written to SFR_CC. The result is written to GPR Rd.

Comments

R-Typ

The value in GPR Rs will be ignored and the shift width will be always one if single shift is configured.
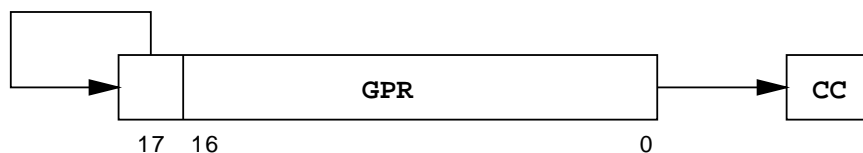
Figure 1-18: shift right arithmetic immediate

trap   trap

trap

Mnemonic

trap   Number

special2	number h	number l	trap
17 13	12 9	8 5	4 0
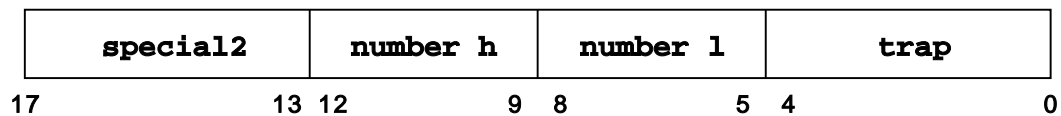
Pseudocode

$RegBase \leftarrow RegBase + 1$

$R11 \leftarrow PC + 1$

$PC \leftarrow SFR\_TR_{17:8} \# \# Number$

Description

This instruction performs a shift of the register window for eight register positions. The current PC is incremented and stored in R11 of the new register window. R11 is used to store the return address of the calling function. The value for RegBase holding the current subroutine call level ($SFR\_STATUS_{6:0}$) is also incremented. Finally, the PC is set to the 10 bit value of $SFR\_TR_{17:8}$ and to the 8 bit value of number (number h ## number l).

Comments

R-Typ

number = number h ## number l

xor

exclusive or

xor

Mnemonic

xor   Rd, Rs

special2	Rd	Rs	xor

```
17          13 12     9 8     5 4          0
```
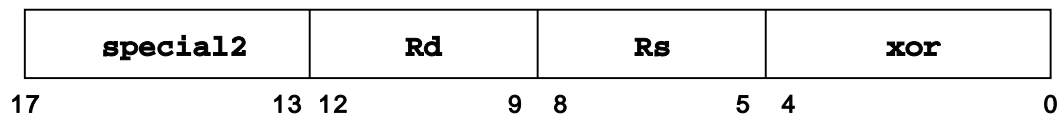
Pseudocode

Rd ← Rd xor Rs

Description

The content of GPR Rd is combined with the content of GPR Rs in a bitwise logical XOR operation, and the result is written to GPR Rd.
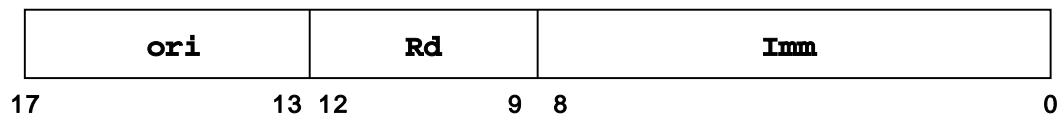
Comments

R-Typ

xori   xori

exclusvie or immediate

Mnemonic

xori   Rd, Imm

xori	Rd	Imm
17 13	12 9	8 0
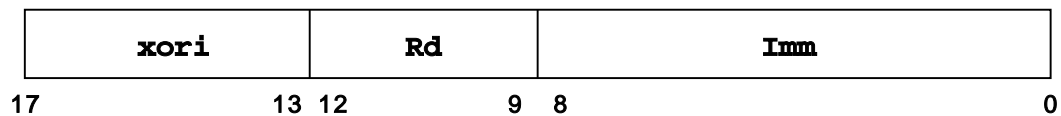
Pseudocode

$Rd \leftarrow Rd \ xor \ 0^9 \ \#\# \ IR_{8:0}$

Description

The zero-extended 9 bit immediate is combined with the content of GPR Rd in a bitwise logical XOR operation, and the result is written to GPR Rd.

Comments

I-Typ

2. Memory Organization

The SpartanMC main memory is a compound of single memory blocks of 2k rows with 18 bit width. The number of blocks and therefore the size of the main memory is configurable. The memory blocks are implemented by using the FPGA internal BlockRAMs. Each block consists of two FPGA BlockRAMs of 2k rows and 9 bit width. Since the FPGA BlockRAMs are dual ported, one port is used to read instructions and the other port is used to read and write data.

The SpartanMC stores data in big endian byte order.
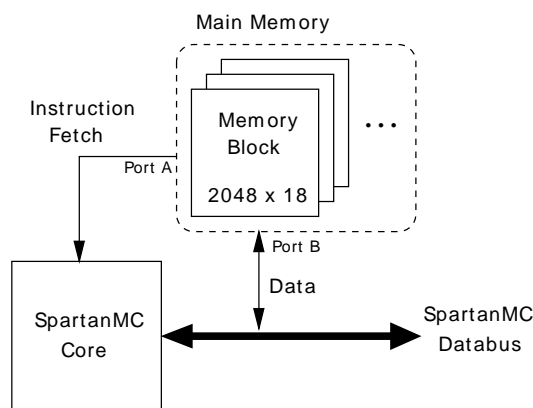
Figure 2-19: Dual ported main memory

2.1. Address Management

Each port of the main memory is connected to a 18 bit address bus. Since the main memory consists of 2k x 18 bit blocks, there are 11 bit required to address the rows within a block. The remaining 7 bit of the address bus are used to select the memory block. Therefore a possible maximum of addressable memory of 256k of 18 bit words distributed to 128 memory blocks could be instantiated. For the instruction port of the memory, the program counter (PC) is used as address bus.

For better memory utilization of the data section the data port provides a 9 bit wise memory access. Therefore the least significant bit (Align) of the data address bus is used to select the upper or lower half word which is used in load and store instructions (l9,s9). The remaining 17 bit are used to address the lower 128k of the memory. To address the upper 128k, the content of SFR_STATUS_7(MM) is used as most significant bit of the data address bus.
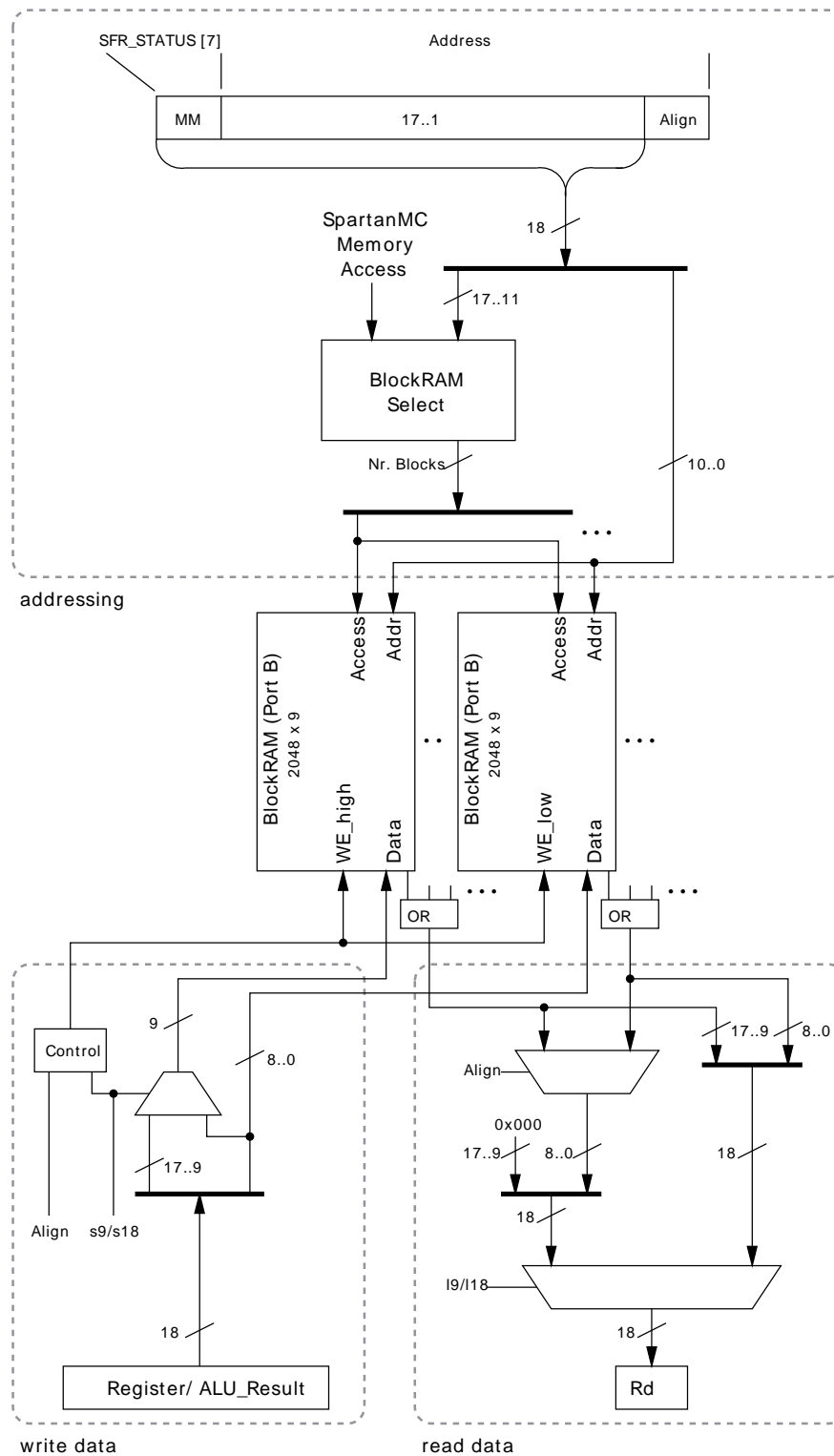
Figure 2-20: Data address management

Note: Due to the 9 bit wise data access, the correct address assignment of data addresses in assembler code has to be assured. The address value of the data address has to be twice the size of the regular instruction address.

2.2. Peripheral Access

2.2.1. Memory Mapped

Peripherals are connected to the regular data and address bus of the SpartanMC. Thus, peripheral devices are mapped to the SpartanMC address space at a dedicated address (IO_BASE_ADR). For exchanging small amounts of data between processor and peripheral, peripherals can provide a set of 18 bit registers. These registers are implemented as distributed memory on the FPGA.
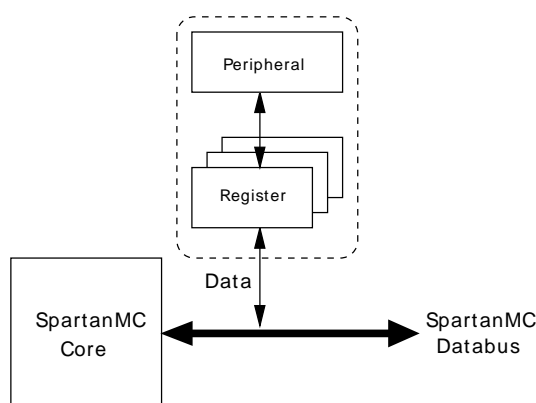
Figure 2-21: Memory mapped registers

The upper 8 bit part of the 18 bit address is used to select the peripheral address space. The selection is carried out by comparing the upper 8 bit part of the current address with the upper 8 bit of the configured base address (IO_BASE_ADR). The lower 10 bit are used to select the peripheral register within this address space. Therefore the 10 bit are divided into two parts: the first 9..n bit to access the correct peripheral module according to the BASE_ADR of the module and the second n-1..0 bit to access the 18 bit register within this peripheral module. The value of n depends on the number of registers provided by the peripheral (e.g. a value of n=3 implies a maximum of 8 registers for that module).

Note: The base address of the peripheral modules should be sorted by the number of registers. Starting with the peripheral using the most registers. This scheme avoids the overlapping of address spaces between different peripherals.

The data access to the registers is similar to the access to the main memory. For reading data (l9/l18) the align bit (LSB of the address) can be used to select the upper or lower half word of the register. For writing data the align bit is meaningless therefore only the s18 operation can be performed on peripheral memory.
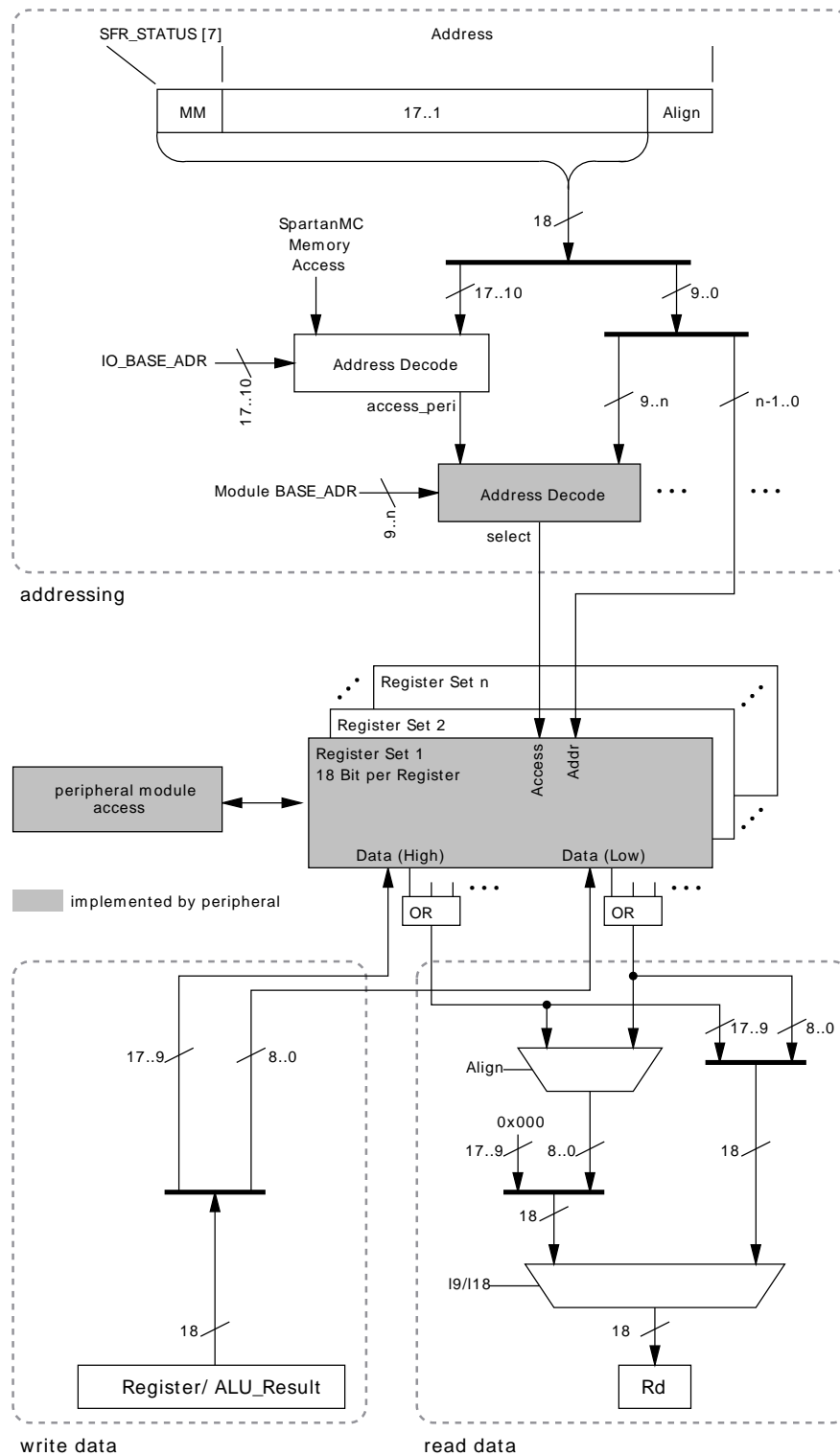
Figure 2-22: Peripheral register address management

2.2.2. Direct Memory Access (DMA)

Peripherals that work on large volumes of data can use BlockRAMs as data interface to the processor. In this case the first port is connected to the SpartanMC address and data bus and the second port is connected to the peripheral which works autonomously on the data in the memory block. This can be regarded as DMA style operation.
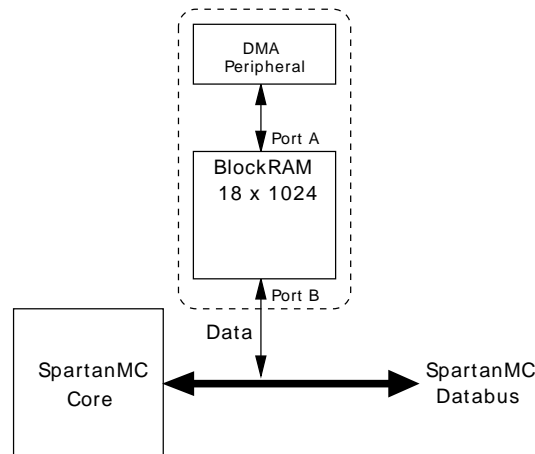
Figure 2-23: DMA with dual ported BlockRAM

Note: Due to the SpartanMC memory management which uses the second port of the BlockRAM as instruction fetch, the processor can not execute code from the DMA memory since the second port is used as peripheral interface. This missing master mode DMA would lead to copying overhead if data needs to be buffered between processing it with different peripherals.

The upper 8 bit part of the 18 bit address is used to select the DMA device. The selection is carried out by comparing the upper 8 bit part of the current address with the upper 8 bit of the configured base address (DMA_BASE_ADR). The lower 10 bit are used to select the row within the DMA BlockRAM.

The data access to the DMA memory is similar to the access to the main memory. For reading data (I9/I18) the align bit (LSB of the address) can be used to select the upper or lower half word of the register. For writing data, the half word to be written is selected by the store_access_low and store_access_high lines.
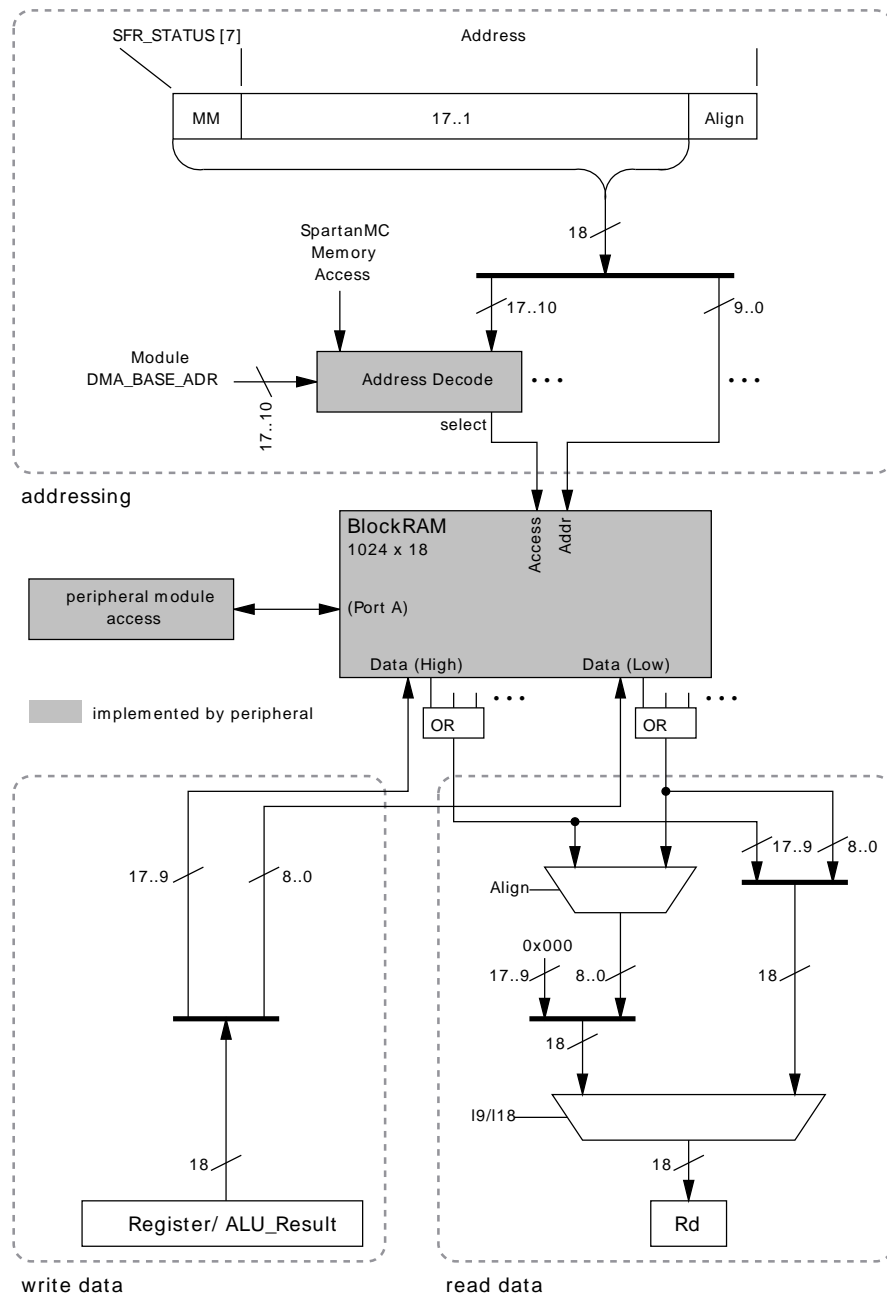
Figure 2-24: DMA address management

2.2.3. Data Read Interface

The main memory blocks and the peripheral memory are connected to the data memory interface of the processor core. In order to avoid tri-state buffers, all incoming data is combined through a wide or-gate. Thus, all memory blocks and peripherals that are currently not addressed must provide a value of zero on their outputs.

2.3. Data and Code Buses

This section describes the various signals making up the code and data buses and their timing.

2.3.1. Data Bus

Table 2-4: Data Bus Signals

Signal	Source Element	Descripton
clk	Core	Clock
reset	Core	Reset Signal
mem_access	Core	Signals that the address is valid and a read / write should be performed
mem_wr_low / mem_wr_high	Core	Write the lower / upper halfword to memory. Only relevant if mem_access is set.
mem_addr_high / mem_addr_block	Core	Memory address in bytes. Only relevant if mem_access is set.
mem_comi	Core	Data to write (Core out, Memory in). Only relevant if mem_access and mem_wr_* are set.
mem_busy	Memory	The memory cannot answer a pending request yet. Processor needs to stall.
mem_cimo	Memory	The data read from memory (Core in, Memory out).
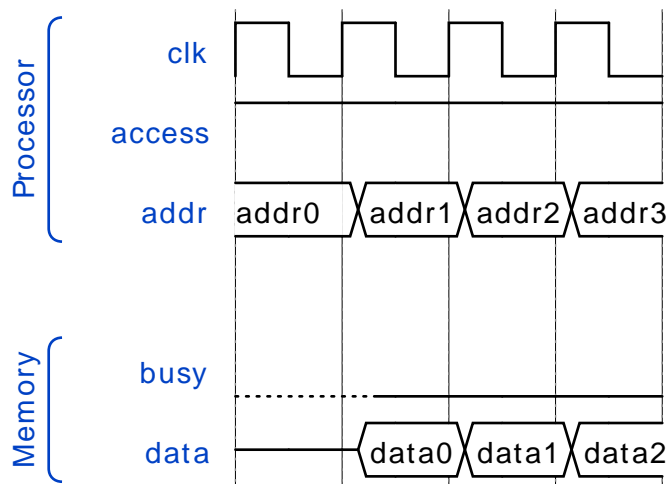
Figure 2-25: Data Bus Access without mem_busy

As shown above, read data needs to be output in the cycle following the request. Data always is read from memory, even if one or both of the write signals are set. The old data is expected to be read from the memory. This is needed for the swap instruction.
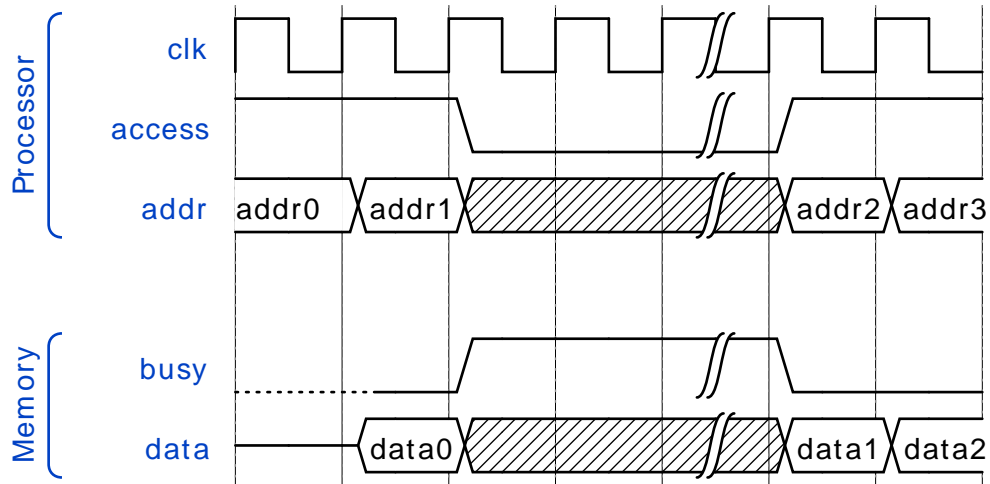
Figure 2-26: Data Bus Access with mem_busy

If a request cannot be served directly, mem_busy needs to be asserted until the request is finished. mem_cimo needs to be driven in the cycle where mem_busy is deasserted. mem_access is guaranteed to not be asserted until mem_busy is deasserted. Therefore, a combinatoric path from mem_busy to mem_access exists inside the processor core. Memory modules therefore must take care not to introduce a combinatoric path from mem_access to mem_busy.

2.3.2. Code Bus

Table 2-5: Data Bus Signals

Signal	Source Element	Descripton
clk	Core	Clock
reset	Core	Reset Signal
code_access	Core	Signals that the address is valid and a read / write should be performed
code_addr	Core	Memory address in words. Only relevant if mem_access is set.
code_jmp	Core	The processor is performing a jump. TODO: When is this signal generated? Before or during jump?
code_busy	Memory	The memory cannot answer a pending request yet. Processor needs to stall.

Signal	Source Element	Descripton
code_cimo	Memory	The code read from memory (Core in, Memory out).
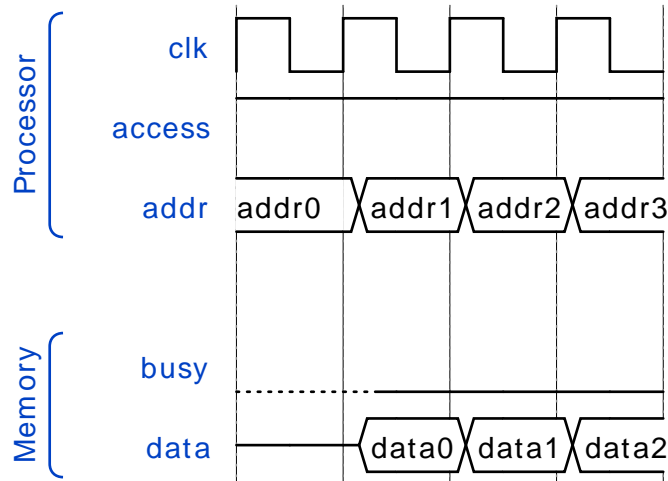
Figure 2-27: Code Bus Access without mem_busy

As shown above, read data needs to be output in the cycle following the request.
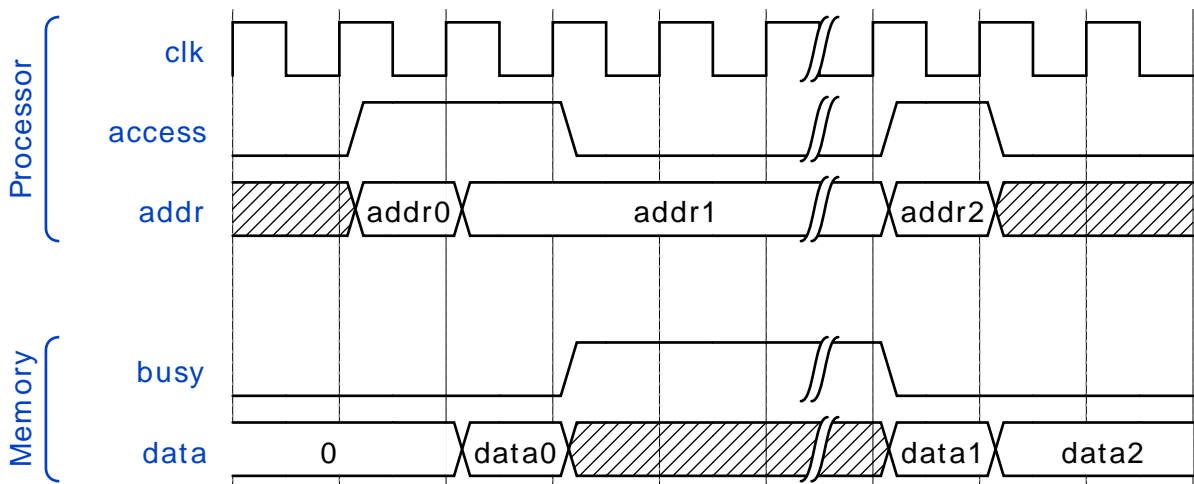
Figure 2-28: Code Bus Access with mem_busy

If a request cannot be served directly, mem_busy needs to be asserted until the request is finished. code_cimo needs to be driven in the cycle where mem_busy is deasserted. code_access is guaranteed to not be asserted until code_busy is deasserted. Therefore, a combinatoric path from code_busy to code_access exists inside the processor core. Memory modules therefore must take care not to introduce a combinatoric path from code_access to code_busy.

While code_access is deasserted, code_cimo needs to keep its previous value. For memories based on block RAMS, this can be achieved by driving its enable input to low.

While code_busy is asserted, the address presented to the code memory stays constant. The memory does not need to output valid data while it is busy.

2.4. Example Memory Map

The following image describes a memory map for an SpartanMC example system and an application using traps and interrupts. The specfic addresses of the different application parts (ISR, Traps, IRQ Handler etc.) are automatically defined through compiler tools. The start addresses of DMA memory (in this example 0x19000) can be defined in the hardware configuraion generated through jConfig.
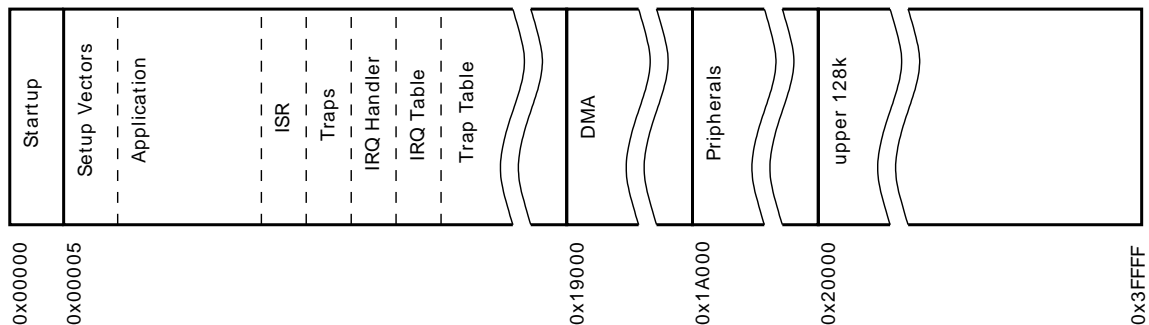
Figure 2-29: Example memory map

Startup: The startup code is generated by compiler tools at address 0x0000. It contains a branch to the application specific *Setup Vectors* -subroutine. The required branch address is defined within the system headers generated via jConfig.

Setup Vectors: Setup the address for the interrupt handler and the trap base address for this application.

Application: Contains the application code.

ISR: The interrupt service routines for the defined interrupts

Traps: The trap code for the defined traps.

IRQ Handler: Performs the IRQ prolog and epilog and links the IRQ table of the application.

IRQ Table: The interrupt branch table. Each 18 bit address contains the jump instructions to the interrupt code. The table length depends on the number of configured interrupts.

Trap Table: The branch table for traps. Each 18 bit address contains the jump instructions to a specific trap code. Since the the upper 10 bit are used as trap base address a maximum of 255 traps can be defined using the lower 8 bit. The implemented table length depends on the number of traps defined in the application

DMA: The memory section for DMA capable peripherals. This memory section is 18 bit aligned and contains data only.

Peripherals: The memory section for memory mapped peripherals. The start address of this section has to be set beyond the actual configured main memory section.

3. Performance counter

The performance counter counts clock cycles and certain events that are generated by the SpartanMC core. Counting can be enabled and disabled by software to profile specific sections of code. The performance counter is controlled by two special function registers. They allow starting and stopping the counters as well as configuring the event type that is counted. All counters provide a configurable prescaler. Event counters can detect edges to count events that last longer than one clock cycle.

3.1. Module Parameters

Table 3-6: Performance counter module parameters

Parameter	Default Value	Descripton
PERFORMANCE_COUNTER	0	Specifies whether the SpartanMC core is equipped with a performance counter.
COUNT_CYCLECOUNTERS	2	Specifies the number of cycle counters.
COUNT_EVENTCOUNTERS	2	Specifies the number of event counters.
WIDTH_CYCLECOUNTER	36	Specifies the width of the cycle counter. Possible values are 18 or 36 bits.
WIDTH_EVENTCOUNTER	36	Specifies the width of the event counters. Possible values are 18 or 36 bits.

3.2. Special function registers

Table 3-7: Performance counter special function registers

SFR name	SFR number	Descripton
sfr_pcnt_idx	9	Used to specify the register that is accessed by writing or reading sfr_pcnt_dat. Also contains global counter enable/disable bits.
sfr_pcnt_dat	10	Writing/reading this register results in a write/read of the register specified by sfr_pcnt_idx.

Access to the performance counter registers is provided by two special function registers: **sfr_pcnt_idx** and **sfr_pcnt_dat.** sfr_pcnt_idx is an index register that specifies which register is read or written when sfr_pcnt_dat is read/written. The layout of

sfr_pcnt_idx is shown in the table below. sfr_pcnt_idx also provides two global enable and disable bits. They can be used to enable/disable all counters with only one instruction.

Table 3-8: sfr_pcnt_idx register layout

Bits	Default value	Descripton
0 - 3	0	Register number.
4 - 7	0	Selected cycle/event counter.
8	0	0: Access cycle counter register. 1: Access event counter register.
9 - 15	0	Unused.
10	0	Writing 1 disables cycle counter 3. Always reads as 0.
11	0	Writing 1 enables cycle counter 3. Always reads as 0.
12	0	Writing 1 disables cycle counter 2. Always reads as 0.
13	0	Writing 1 enables cycle counter 2. Always reads as 0.
14	0	Writing 1 disables cycle counter 1. Always reads as 0.
15	0	Writing 1 enables cycle counter 1. Always reads as 0.
16	0	Writing 1 disables cycle counter 0. Always reads as 0.
17	0	Writing 1 enables cycle counter 0. Always reads as 0.

3.3. Performance counter registers

The cycle counter and every event counter has three registers, one configuration register and two registers that contain the counter value. Registers are selected by setting **sfr_pcnt_idx** according to table 3-6. After that the registers can be read or written by reading or writing to **sfr_pcnt_dat** .

Table 3-9: Performance counter registers

Register number	Descripton
0x00	Cycle/event counter configuration register.
0x01	Cycle/event counter value (bits 0-17).
0x02	Cycle/event counter value (bits 18-35).

The cycle counter configuration register is used to configure the cycle counters. It also contains information about the amount of available cycle and event counters.

Table 3-10: Cycle counter configuration register layout

Bits	Default value	Descripton
0	0	Enable counter. 0: Counter disabled. 1: Counter enabled.
1	0	Counter reset. Writing 1 resets the counter to zero. Always reads as 0.
2	0	Interrupt filter enable. Writing 1 disables counting while the processor handles an interrupt.
4-7	0	Prescaler value: 0: No prescaler 1: Prescaler 2 2: Prescaler 4 3: Prescaler 8 ... 15: Prescaler 32768
8	0	Cycle counter overflow flag. Writing 1 will clear the flag.
9-11	1	Number of cycle counters.
12-16	2	Number of event counters.

The event counter configuration register defines the behaviour of the event counters. It is used to set the event that is counted as well as enabling edge detection or a prescaler.

Event counters can be tied to a cycle counter. This is done by setting bit 6 to 1 and bits [4:5] to the number of the cycle counter. When an event counter is tied to a cycle counter it uses the cycle counters enable and reset signals. This means that the event counter is only counting events when the cycle counter is running. Enabling the interrupt filter for a cycle counter will also enable it for all event counters that are tied to that cycle counter.

Table 3-11: Event counter configuration register layout

Bits	Default value	Descripton
0-3	0	Selected event number.
4-5	0	Cycle counter that the event counter is tied to.
6	0	Tie event counter to a cycle cunter.
7	0	Enable event counter. 0: Counter disabled. 1: Counter enabled.
8	0	Reset event counter. Always reads as 0.
9	0	Writing 1 enables the interrupt filter.

Bits	Default value	Descripton
10	0	Edge detect. 1 enables edge detection.
11	0	Event counter overflow flag. Writing 1 will clear the flag.
12-15	0	Prescaler value: 0: No prescaler 1: Prescaler 2 2: Prescaler 4 3: Prescaler 8 ... 15: Prescaler 32768

3.4. Countable events

Table 3-10 lists all countable events. The mnemonics are defined in **perf.h** .

Table 3-12: Countable events

Event number	Mnemonic	Descripton
0	EVENT_STALL	Pipeline stall.
4	EVENT_DWRITE	Data write.
5	EVENT_DREAD	Data read.
6	EVENT_INTR	Interrupt generated.
7	EVENT_NOP	nop instruction executed.
8	EVENT_EXT1	External event line 1.
9	EVENT_EXT0	External event line 0.
10	EVENT_CCNT0_OVF	Cycle counter 0 overflow.
11	EVENT_CCNT1_OVF	Cycle counter 1 overflow.
12	EVENT_CCNT2_OVF	Cycle counter 2 overflow.
13	EVENT_CCNT3_OVF	Cycle counter 3 overflow.

3.5. Example code

The following example code demonstrates the use of the performance counters. In this example a function **do_work(uint36_t i)** is profiled. This function generates **i** read and write accesses to memory. In addition to that an interrupt is generated every 256 clock cycles using a RTI timer. After profiling has finished the result is printed over a UART interface.

The example uses a performance counter with one cycle counter and two event counters. The event counters are tied to the cycle counter.

```c
#include <peripherals.h>
#include <moduleParameters.h>
#include <perf.h>
#include <interrupt.h>
#include <stdio.h>

static void do_work(uint36_t i)
{
   volatile int a = 0, b = 0;

   while(i--) {
      b = a;
      asm volatile("nop");
   }

   return;
}

   FILE * stdout = &spartanmc_0_uart_light_0_file;

int main(void)
{
   int i;
   struct perf_conf c;
   struct perf_result r;

   printf("Started!\r\n");

   c.num_ccnt = 1;
   c.num_evcnt = 2;
   c.cycle[0].prescaler = 0;
   c.cycle[0].intr_filter = 1;

   c.event[0].eventnum = EVENT_DREAD;
   c.event[0].prescaler = 0;
   c.event[0].edge_detect = 1;
   c.event[0].tie_ccnt = 1;
   c.event[0].tied_ccnt = 0;

   c.event[1].eventnum = EVENT_INTR;
   c.event[1].prescaler = 0;
   c.event[1].edge_detect = 1;
   c.event[1].tie_ccnt = 1;
   c.event[1].tied_ccnt = 0;
   perf_init(&c);
```

```
    spartanmc_0_rti_0.ctrl = 0x23; // RTI prescaler 256
    interrupt_enable();

    printf("Profiling started: Interrupts enabled.\r\n");
    perf_start(0);
    do_work(1000);
    perf_stop(0);
    perf_read(&r);
    perf_results_printf(&r);
    perf_reset();

    printf("Profiling started: Interrupts disabled.\r\n");
    spartanmc_0_rti_0.ctrl = 1;
    perf_start(0);
    do_work(1000);
    perf_stop(0);
    perf_read(&r);
    perf_results_printf(&r);
    perf_reset();

    printf("Entering endless loop.\r\n\n\n");
    while(1) {
        ;
    }

    return 0;
}

void isr00(void)
{
    // Reset interrupt flag.
    volatile int t = spartanmc_0_rti_0.ctrl;

    return;
}
```

The results received via UART look like this:

```
Started!
Profiling started: Interrupts enabled.
========== Performance counter results ==========
Cycle0: 9086
Event0: 1000
Event1: 39

Profiling started: Interrupts disabled.
========== Performance counter results ==========
```

```
Cycle0: 9007
Event0: 1000
Event1: 0

Entering endless loop.
```

3.6. perf.h header file

```
#include <stdint.h>

#define MAX_CYCLECOUNTERS    4
#define MAX_EVENTCOUNTERS   16

#define EVENT_CCNT3_OVF    13
#define EVENT_CCNT2_OVF    12
#define EVENT_CCNT1_OVF    11
#define EVENT_CCNT0_OVF    10
#define EVENT_EXT0          9
#define EVENT_EXT1          8
#define EVENT_NOP           7
#define EVENT_INTR          6
#define EVENT_DREAD         5
#define EVENT_DWRITE        4
#define EVENT_IREAD         3
#define EVENT_IHIT          2
#define EVENT_IMISS         1
#define EVENT_STALL         0

struct evcnt_conf {
   uint9_t prescaler; // Prescaler value.
   uint9_t eventnum; // Number of the counted event.
   uint9_t tied_ccnt;
   uint9_t edge_detect; // Edge detection enabled?
   uint9_t intr_filter; // Interrupt filtering enabled?
   uint9_t tie_ccnt;
};

struct ccnt_conf {
   uint9_t prescaler;
   uint9_t intr_filter;
};


struct perf_conf {
   uint9_t num_ccnt;
   uint9_t num_evcnt;
```

```
   // Configuration for the cycle counters.
   struct ccnt_conf cycle[MAX_CYCLECOUNTERS];
   // Configuration for the event counters.
   struct evcnt_conf event[MAX_EVENTCOUNTERS];
};

struct perf_result {
   struct {
      uint36_t counter;
      uint9_t overflow;
   } cycles[MAX_CYCLECOUNTERS];

   struct {
      uint36_t counter;
      uint9_t overflow;
   } events[MAX_EVENTCOUNTERS];
};

// Used by AutoPerf.
struct perf_auto_result {
   char *name;
   struct perf_result result;
};

// Sets the global enable bit of cycle counter a to 1.
#define perf_start(a)   asm volatile("movi2s sfr_pcnt_idx,
%0" : : "r"(1<<(17 - 2*(a)) ))
// Sets the global enable bit of cycle counter a to 0.
#define perf_stop(a)   asm volatile("movi2s sfr_pcnt_idx,
%0" : : "r"(1<<(16 - 2*(a)) ))

// Sets the configuration registers according to conf.
int perf_init(struct perf_conf *conf);
// Resets all counters and overflow flags.
void perf_reset(void);
// Reset a specific cycle counter.
void perf_reset_cycle(int n);
// Reset a specific event counter.
void perf_reset_event(int n);
// Reads the results and stores them in res.
void perf_read(struct perf_result *res);
// Uses printf() to print the results.
void perf_results_printf(struct perf_result *res);

// Used by AutoPerf.
#define perf_auto_start() perf_start()
```

```
#define perf_auto_stop(/*int*/a,/*struct perf_auto_result* */
b)\
   do {\
      perf_stop();\
      perf_auto_stop_((a),(b));\
   } while(0)

void perf_auto_init(void);
void perf_auto_stop_(int n, struct perf_auto_result *r);
void perf_auto_print(int size, struct perf_auto_result *r);
```

4. Simple Interrupt Controller (IRQ-Ctrl)

Depending on the requirements of the target application two types of interrupt controllers could be instantiated, IRQ-Ctrl and IRQ-Ctrlp. The simple interrupt controller (IRQ-Ctrl) provides a small resource footprint, but handles only one interrupt at once. Incoming interrupts (even of higher priority) will be ignored during the interrupt handling until the Interrupt Service Routine (ISR) is completed. Thus, the running ISR execution is **not** interruptable.
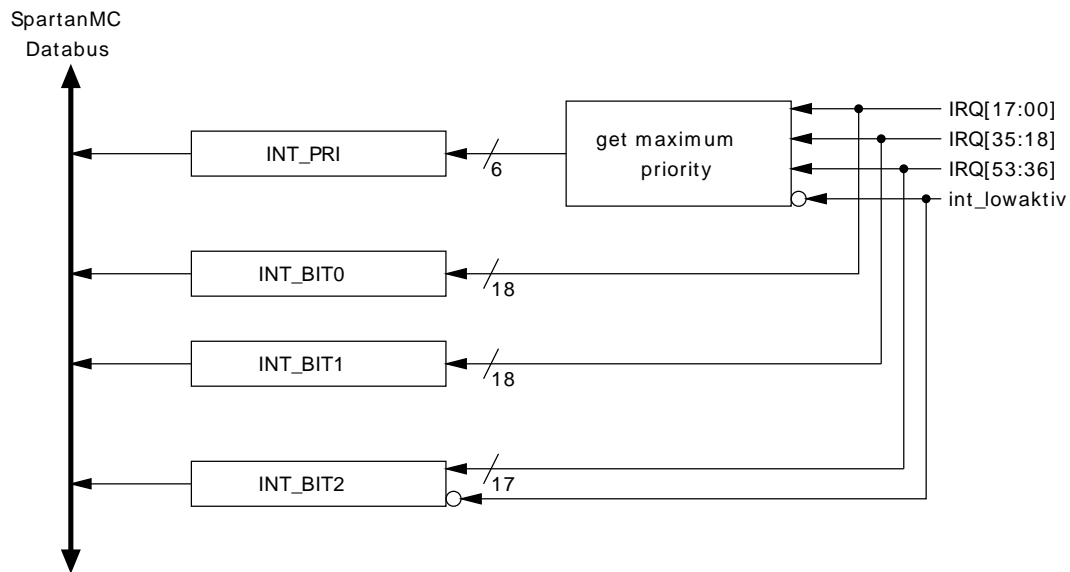
Figure 4-30: IRQ-Ctrl block diagram for IR_SOURCES=54

4.1. Function

The SpartanMC interrupt controller handles multiple interrupt sources as input sorted by their prority. Each interrupt capable peripheral must use a dedicate controller input signal for its interrupt request. If an interrupt occures the controller sets the interrupt input signal of the pipeline. The interrupt number of the pending interrupt with the highest priority could be read from INT_PRI register. In order to check all interrupt sources, the controller provides a configurable number of 18 bit registers (INT_BIT0..2) each with 18 interrupt flags.

Pending interrups are **not** stored within the interrupt controller. Thus, the logic to set/hold, reset and mask interrups must be provided by the peripheral which is connected to the interrupt controller.

Note: The highest interrupt priority is reserved for the "int_lowactive" signal.

4.2. Module parameters

Table 4-13: IRQ-Ctrl modul parameters

Parameter	Default Value	Descripton
BASE_ADR	0x40	Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
IR_SOURCES	8	Number of IRQ Sources fot the SpartanMC core.

4.3. Peripheral Registers

4.3.1. IRQ-Ctrl Register Description

The IRQ-Ctrl peripheral provides three 18 bit registers for 18 interrupt sources. Register four and five could be used if additional interrupts are required. The registers are mapped to the SpartanMC address space located at 0x1A000 + BASE_ADR + Offset. Register three and four could be used if additional interrupts are required.

Table 4-14: IRQ-Ctrl registers

Offset	Name	Access	Description
0	INT_PRI	read	Register for the current max. Priority IRQ number.
1	INT_BIT0	read	Contains the current IRQ-sinals 0 to 17.
2	INT_BIT1	read	Contains the current IRQ-sinals 18 to 35.
3	INT_BIT2	read	Contains the current IRQ-sinals 36 to 54.

4.3.2. IRQ-Ctrl C-Header for Register Description

```c
#ifndef __INTCTRL_H
#define __INTCTRL_H

#ifdef __cplusplus
extern "C" {
#endif
// Number of interrupts (i_bits) is set by jconfig

typedef struct {
    volatile unsigned int   int_pri;    // read
    volatile unsigned int   int_bit0;   // read      17:00
    volatile unsigned int   int_bit1;   // read      35:18
    volatile unsigned int   int_bit2;   // read      53:36
} intctrl_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/intctrl.h

4.4. Firmware

4.4.1. Changes to config-build.mk

Add `interrupt` to the list of `LIB_OBJ_FILES`.

4.4.2. Interrupt Service Routine

The declaration of the interrupt service routines must have the following pattern:

```
ISR(Number or the ISR)()
```

Interrupts need to be enabled globally before they can be used in the system. This is done by calling `interrupt_enable()`. Note that the function is declared in the header file `interrupt.h`.

5. Complex Interrupt Controller (IRQ-Ctrlp)

Depending on the requirements of the target application two types of interrupt controllers could be instantiated (IRQ-Ctrl and IRQ-Ctrlp). The complex interrupt controller (IRQ-Ctrlp) allows the interruption of a running Interrupt Service Routine (ISR) by an interrupt of higher priority. Therefore, the interrupt enable bit (function call of "interrupt_enable()") must be set within the ISR. The complex interrupt controller can handle a maximum of 8 nested interrupts. The 9'th nested interrupt invocation is executed after completion of the 8'th ISR. It should be noted, that IRQ-Ctrlp requires much more FPGA resources than the simple IRQ-Ctrl.
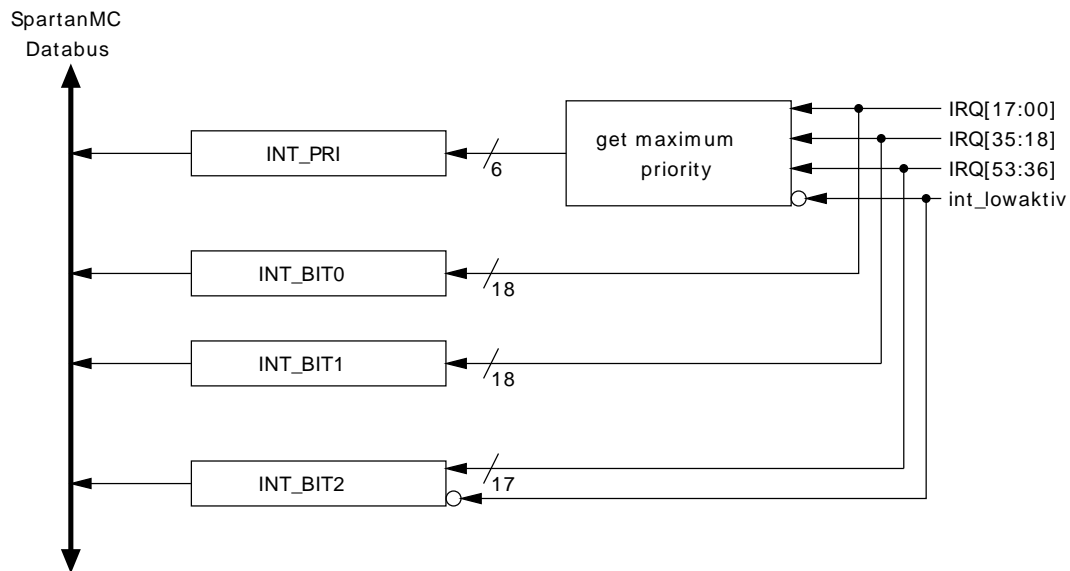
Figure 5-31: IRQ-Ctrl block diagram for IR_SOURCES=54

5.1. Function

The SpartanMC interrupt controller handles multiple interrupt sources as input sorted by their prority. Each interrupt capable peripheral must use a dedicate controller input signal for its interrupt request. If an interrupt occures the controller sets the interrupt input signal of the pipeline. The interrupt number of the pending interrupt with the highest priority could be read from INT_PRI register. In order to check all interrupt sources, the controller provides a configurable number of 18 bit registers (INT_BIT0..2) each with 18 interrupt flags.

Pending interrups are **not** stored within the interrupt controller. Thus, the logic to set/ hold, reset and mask interrups must be provided by the peripheral which is connected to the interrupt controller.

Note: The highest interrupt priority is reserved for the "int_lowactive" signal.

5.2. Module parameters

Table 5-15: IRQ-Ctrl modul parameters

Parameter	Default Value	Descripton
BASE_ADR	0x40	Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
IR_SOURCES	8	Number of IRQ Sources at SpartanMC Core.

5.3. Peripheral Registers

5.3.1. IRQ-Ctrl Register Description

The IRQ-Ctrl peripheral provides three or more 18 bit registers which are mapped to the SpartanMC address space located at 0x1A000 + BASE_ADR + Offset.

Table 5-16: IRQ-Ctrl registers

Offset	Name	Access	Description
0	INT_PRI	read	Register for the current max. Priority IRQ-Number.
1	INT_BIT0	read	Contains the current IRQ-Sinals 0 to 17.
2	INT_BIT1	read	Contains the current IRQ-Sinals 18 to 35.
3	INT_BIT2	read	Contains the current IRQ-Sinals 36 to 54.

5.3.2. IRQ-Ctrl C-Header for Register Description

```
#ifndef __INTCTRLP_H
#define __INTCTRLP_H

#ifdef __cplusplus
extern "C" {
#endif
// Number of interrupts (i_bits) is set by jconfig

typedef struct {
   volatile unsigned int   int_pri;   // read
   volatile unsigned int   int_bit0;  // read      17:00
   volatile unsigned int   int_bit1;  // read      35:18
   volatile unsigned int   int_bit2;  // read      53:36
} intctrlp_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/intctrlp.h

5.4. Firmware

5.4.1. Changes to config-build.mk

Add `interrupt` to the list of `LIB_OBJ_FILES`.

5.4.2. Interrupt Service Routine

The declaration of the interrupt service routines must have the following pattern:

```
ISR(Number or the ISR)()
```

Interrupts need to be enabled globally before they can be used in the system. This is done by calling `interrupt_enable()`. Note that the function is declared in the header file `interrupt.h`.

6. Universal Asynchronous Receiver Transmitter (UART)

The UART is a SpartanMC peripheral device for serial communication with external systems. The UART enables a bit stream of 5-8 bits to be shifted in and out of the peripheral at a programmed bit rate. The peripheral is connected with the external system environment by the two signals Rx and Tx. If the UART is parametrized as modem, the additional signals DTR, DCD and CTS are usable. The incoming and outgoing data is written to configurable FIFO memory modules which are connected to the input and output shift registers of the UART.
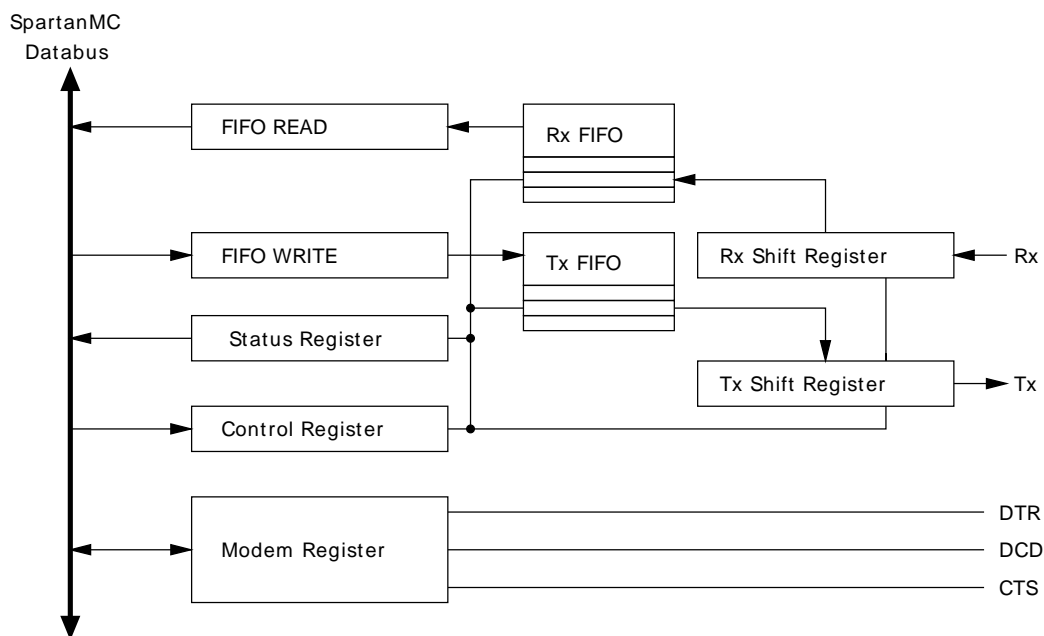
Figure 6-32: UART block diagram

6.1. Framing

Each frame starts with a logic low start bit followed by a configurable number of data bits (5-8), an optional parity bit and one or more logic hight stop bits. The parity bit can be defined as odd or even. If the parity is set to "even", the hamming weight of all data bits including the partiy bit have to be even for a valid frame. Contrariwise, the hamming weight of all data bits and the parity bit have to be odd if the parity bit is set to "odd". The transmission of the data field starts with the least significant bit (LSB).
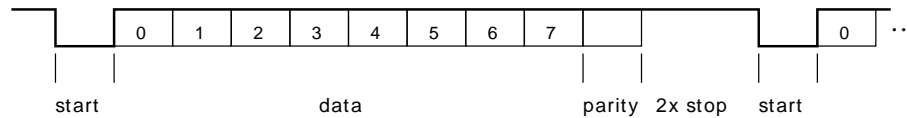
Figure 6-33: UART frame example

6.2. Module parameters

Table 6-17: UART module parameters

Parameter	Default Value	Descripton
BASE_ADR	0x10	Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
FIFO_RX_DEPTH	8	Number of 8 bit cells of the receiver FIFO memory for incoming data.
FIFO_TX_DEPTH	8	Number of 8 bit cells of the sender FIFO memory for outgoing data.
MODEM	0	Allows the usage of the UART peripheral control signals. If MODEM is set to one the control signals RTS, DCD and DTR are active. If MODEM is set to zero RTS, DTR and DTR are set to logic high.
CLKIN_FREQ	25000000	The clock frequency in Hz which the module is currently driven by. This frequency is used to calculate the prescalers for the different baud rates available in UART_CTRL.

6.3. Interrupts

For interrupt driven serial communication the UART provides two interrupt signals which will be generated at the following conditions:

- TX interrupt is set for each sent character. The interrupt remains one until the next read access to UART_CTRL (Register Nr. 3).

- RX interrupt is set for each received character. The interrupt remains one until the next read acces to UART_FIFO_READ (Register Nr. 1).

Note: In case the software application uses these interrupts, the SpartanMC SoC requires an interrupt controller which provides an appropriate inferface for the interrupt signals.

6.4. Peripheral Registers

6.4.1. UART Register Description

The UART peripheral provides four 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 6-18: UART registers

Offset	Name	Access	Description
0	UART_STATUS	read	Contains the current Rx/Tx FIFO status, current framing/parity errors, the modem signals if available and the the busy signals for the Rx/Tx shift registers.
1	UART_FIFO_READ	read	Register for incoming data. The LSB of the data word is written to $UART\_FIFO\_READ_0$.
2	UART_FIFO_WRITE	write	Register for outgoing data. The LSB of the data word is written $UART\_FIFO\_WRITE_0$.
3	UART_CTRL	read/write	Contains the current UART setting e.g. baud rate, data field length and parity and interrupt settings.
4	UART_MODEM	read/write	Contains the input/output setting for the modem signals.

6.4.2. UART_Status Register

Table 6-19: UART status register layout

Bit	Name	Access	Default	Description
0	RX_EMPTY	read	1	Set to one if the Rx FIFO is empty otherwise the value is zero.
1	RX_FULL	read	0	Set to one if the Rx FIFO is full otherwise the value is zero.
2	TX_EMPTY	read	1	Set to one if the Tx FIFO is empty otherwise the value is zero.
3	TX_FULL	read	0	Set to one if the Tx FIFO is full otherwise the value is zero.
4	TX_IRQ_PRE	read	1	Set to one while data were witten to the Tx FIFO. The value is reset to zero through the next TX access to the Tx FIFO memory.
5	TX_IRQ_FLAG	read	0	Set to one while data were sent. The value is reset to zero through the next write access to TX_IRQ_FLAG.
6	RX_P_ERR	read	0	Set to one until the next frame if a parity error occurs.
7	RX_F_ERR	read	0	Set to one until the next frame if a frameing error (break character or high impetance input) occurs.
8	RX_D_ERR	read	0	Set to one if the Rx FIFO is full while receiving a frame. The value remains one until the status of the Rx FIFO is not full.
9	M_DCD	read	0	The current value of the modem signal DCD if the parameter MODEM is set to one.
10	M_CTS	read	0	The current value of the modem signal CTS if the parameter MODEM is set to one.
11	M_DSR	read	0	The current value of the modem signal DSR if the parameter MODEM is set to one.
12	RX_CLK	read	0	Current clock of the Rx shift register. (For debugging purposes)
13	RX_STOP	read	0	It is set to one while the Rx shift register is not busy.
14	TX_CLK	read	0	Current clock of the Tx shift register. (For debugging purposes)
15	RST_UART	read	0	Is set to one during a UART reset. Typically, the reset is started with the SpartanMC processor core reset signal. The reset remains one for one UART bit period. This is equivalent to 8.6 us at 115200 Baud.
16	TX_STOP	read	0	It is set to one while the Tx shift register is not busy.
17	x	read	0	Not used.

Table 6-19: UART status register layout

6.4.3. UART_FIFO_READ Register

Table 6-20: UART status register layout

Bit	Name	Access	Default	Description
0-7	RX	read	x	Register for received data.
8-17	x	read	0	Not used.

6.4.4. UART_FIFO_WRITE Register

Table 6-21: UART status register layout

Bit	Name	Access	Default	Description
0-7	TX	write	x	Register for data to send.
8-17	x	write	x	Not used.

6.4.5. UART_CTRL Register

Note: Befor writing UART_CTRL it has to be assured that RST_UART are set to zero and RX_EMPTY, TX_EMPTY, RX_STOP and TX_STOP are set to one.

Table 6-22: UART control register layout

Bit	Name	Access	Default	Description
0	RX_EN	write	0	Turns the Rx shift register off if set to zero.
1	TX_EN	write	0	Turns the Tx shift register off if set to zero.
2	PARI_EN	write	0	If set to one the usage of parity bits in frames will be enabled.
3	PARI_EVEN	write	0	If set to one the parity is even otherwise the parity is odd.
4	TWO_STOP	write	0	Enables the usage of a second stop bit.
5-7	DATA_LEN	write	000	Sets the length of the data field : 111 = 8 bit data 110 = 7 bit data 101 = 6 bit data 100 = 5 bit data
8	x	write	x	Not used.
9-12	BPS	write	0000	Sets the baud rate: 0000 = 115200 Baud 0001 = 57600 Baud 0010 = 38400 Baud 0011 = 31250 Baud (MIDI data rate) 0100 = 19200 Baud 0101 = 9600 Baud 0110 = 4800 Baud 0111 = 2400 Baud 1000 = 1200 Baud 1001 = 600 Baud 1010 = 300 Baud 1011 = 150 Baud 1100 = 75 Baud 1101 = 50 Baud All other values are mapped to 7812.5 Baud.
13	RX_IE	write	0	If set to one the Rx interrupt will be enabled.

Bit	Name	Access	Default	Description
14	TX_IE	write	0	If set to one the Tx interrupt will be enabled.
15	TX_BREAK	write	0	If set to one the UART will sent a break signal. (Tx logic low) The duration of the break signal must be longer than a complete frame (start, data, stop and parity). To identify breaks the framing error detection can be used.
16-17	x	write	x	Not used.

Table 6-22: UART control register layout

6.4.6. UART_MODEM Register

The UART peripheral provides three control lines for hardware handshaking and flow control: Data Carrier Detect (DCD), Data Set Ready (DSR)/Data Terminal Ready (DTR) and Request To Send (RTS)/Clear TO Send (CTS). The signal name and the i/o-direction depends on the RS-232 device class - Data Terminal Equipment (DTE) or Data Communication Equipment (DCE). For more information cf. the EIA-232 or RS-232 standard.

Table 6-23: UART modem register layout

Bit	Name	Access	Default	Description
0	DTR_DSR	read/ write	0	If DTR is used as output DTR tells DCE that DTE is ready to be connected. If DTR is used as input DSR tells DTE that DCE is ready to receive commands or data.
1	RTS_CTS	write	0	If RTS is used as input RTS tells DCE to prepare to accept data from DTE. If RTS is used as output CTS acknowledges RTS and allows DTE to transmit.
2	DCD	write	0	Tells the DTE that DCE is connected to the carrier line.
3	DCD_OUT	write	0	If set to one DCD works as output which indicates the peripheral is used as DCE. If set to zero DCD works as input and the peripheral is used as DTE.
4	RTS_OUT	write	0	If set to one RTS works as output which indicates the peripheral is used as DTE. If set to zero RTS works as input and the peripheral is used as DCE.
5	DTR_OUT	write	0	If set to one DTR works as output which indicates the peripheral is used as DTE. If set to zero DTR works as input and the peripheral is used as DCE.
6-17	x	write	x	Not used.

Table 6-23: UART modem register layout

6.4.7. UART C-Header for Register Description

```c
#ifndef __UART_H
#define __UART_H

#ifdef __cplusplus
extern "C" {
#endif

#include <stdint.h>

// Status Signale
#define UART_RX_EMPTY   (1<<0)
#define UART_RX_FULL    (1<<1)
#define UART_TX_EMPTY   (1<<2)
#define UART_TX_FULL    (1<<3)
#define UART_TX_IRQ_PRE (1<<4)
#define UART_TX_IRQ_FLAG (1<<5)
#define UART_RX_P_ERR   (1<<6)
#define UART_RX_F_ERR   (1<<7)
#define UART_RX_D_ERR   (1<<8)
#define UART_M_DCD   (1<<9)
#define UART_M_CTS   (1<<10)
#define UART_M_DSR   (1<<11)
#define UART_RX_CLK   (1<<12)
#define UART_RX_STOP   (1<<13)
#define UART_TX_CLK   (1<<14)
#define UART_RST_UART   (1<<15)   // UART noch im RESET, wenn =
1
#define UART_TX_STOP   (1<<16)


// Steuersignale
#define UART_RX_EN   (1<<0)
#define UART_TX_EN   (1<<1)
#define UART_PARI_EN   (1<<2)
#define UART_PARI_EVEN (1<<3)   // 0 = ungerade / 1 = gerade
#define UART_TWO_STOP   (1<<4)   // 0 = ein / 1 = zwei Stopbits

#define UART_DATA_LEN_5 (4<<5)   // 0x00080
#define UART_DATA_LEN_6 (5<<5)   // 0x000A0
#define UART_DATA_LEN_7 (6<<5)   // 0x000C0
#define UART_DATA_LEN_8 (7<<5)   // 0x000E0

#define UART_BPS_115200 (0<<9)   // 0x00000
#define UART_BPS_57600 (1<<9)    // 0x00200
```

```
#define UART_BPS_38400 (2<<9)    // 0x00400
#define UART_BPS_31250 (3<<9)    // 0x00600 MIDI Datenrate
#define UART_BPS_19200 (4<<9)    // 0x00800
#define UART_BPS_9600   (5<<9)    // 0x00A00
#define UART_BPS_4800   (6<<9)    // 0x00C00
#define UART_BPS_2400   (7<<9)    // 0x00E00
#define UART_BPS_1200   (8<<9)    // 0x01000
#define UART_BPS_600    (9<<9)   // 0x01200
#define UART_BPS_300    (10<<9)    // 0x01400
#define UART_BPS_150    (11<<9)    // 0x01600
#define UART_BPS_75    (12<<9)   // 0x01800
#define UART_BPS_50    (13<<9)   // 0x01A00
#define UART_BPS_7812   (14<<9)    // 0x01C00 Boot 68hc11 mit
7812,5 Baud

#define UART_RX_IE    (1<<13)
#define UART_TX_IE    (1<<14)
#define UART_TX_BREAK    (1<<15)

// Modem Outputs und Richtung (optional)
#define UART_DTR_DSR    (1<<0)
#define UART_RTS_CTS    (1<<1)
#define UART_DCD        (1<<2)
#define UART_DCD_OUT    (1<<3)
#define UART_RTS_OUT    (1<<4)
#define UART_DTR_OUT    (1<<5)

typedef struct {
    volatile uint18_t status;    // read
    volatile uint18_t rx_data;    // read (Reset Rx Interrupt)
    volatile uint18_t tx_data;    // write
    volatile uint18_t ctrl_stat; // write (or read = status &
Reset Tx Interrupt)
    volatile uint18_t modem;     // write (optional)
} uart_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```

Code included from spartanmc/include/peripherals/uart.h

7. Simple Universal Asynchronous Receiver Transmitter (UART Light)

The UART Light is a SpartanMC peripheral device for serial communication with external systems. Compared to the standard UART the UART Light uses a lightwight interface which enables a smaller resource footprint. To provide the minimum interface the peripheral is primarily configured via pre synthesis parameters.

The UART Light bitstream is fixed to 8 databits per frame, the datarate and FIFO buffer depth is configurable via module paramater. The signals Rx and Tx are used as connection to the external system environment. The incoming and outgoing data is written to FIFO memory modules which are connected to the input and output shift registers of the UART.
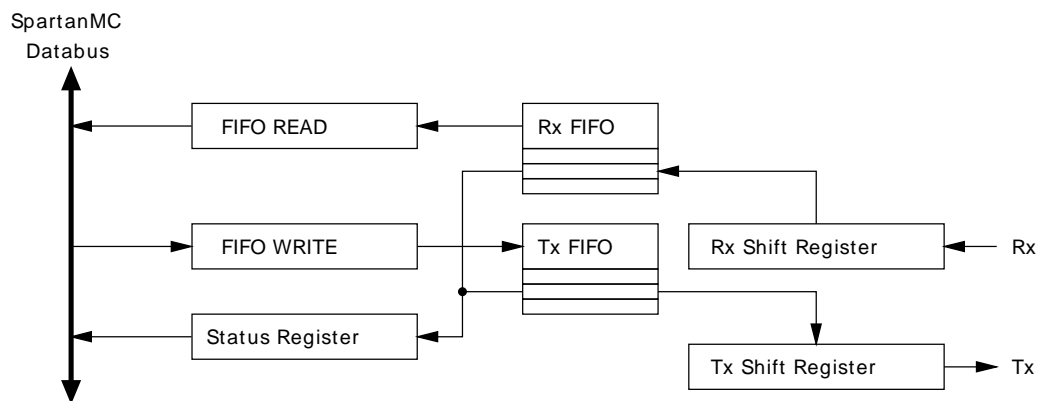
Figure 7-34: UART Light block diagram

7.1. Framing

Each frame starts with a logic low start bit followed by a fixed number of 8 databits. Parity bits and additional stop bits are not supported. The transmission of the data field starts with the least significant bit (LSB).
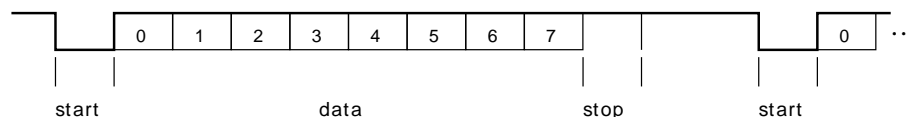
Figure 7-35: UART Light frame

7.2. Module parameters

Table 7-24: UART module parameters

Parameter	Default Value	Descripton
BASE_ADR	0x10	Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
FIFO_RX_DEPTH	8	Number of 8 bit cells for incoming data of the receiver FIFO memory. (maximum 2048)
FIFO_TX_DEPTH	8	Number of 8 bit cells for outgoing data of the sender FIFO memory. (maximum 2048)
BAUDRATE	115200	Defines the required baud rate: 921600 Baud 576000 Baud 460800 Baud 230400 Baud 115200 Baud 57600 Baud 28000 Baud 14400 Baud 7200 Baud 4800 Baud 2400 Baud 1200 Baud 600 Baud 300 Baud
INTERRUPT_SUPPORTED	FALSE	If this parameter is set to FALSE, the content of the FIFO memories must be determined via polling. If this parameter is set to TRUE, each received or sent frame can generates an interrupt.
CLOCK_FREQUENCY	25000000	The clock frequency in Hz which the module is currently driven by. This frequency is used to calculate the prescalers for the configured baudrate.

7.3. Interrupts

In case the UART Light is synthezied for interrupt mode it provides two interrupt signals. Both interrupt signals must be enabled in the UART STATUS register. The interrupts will be generated at the following conditions:

- The Tx interrupt (intr_tx) is set to one if the Tx FIFO status not full and write in last time to Tx-Data. The interrupt indicates that one data in the Tx FIFO memory were sent.

- The Rx interrupt (intr_rx) is set if the first frame is written to the Rx FIFO memory.

The Tx interrupt reset is perfomed by reading the UART_STATUS register. The Rx interrupt reset is perfomed by reading data from the Rx FIFO memory.

Note: In case the software application uses this interrupt, the SpartanMC SoC requires an interrupt controller which provides an appropriate inferface for the interrupt signal.

7.4. Peripheral Registers

7.4.1. UART Register Description

The UART peripheral provides three 18 bit registers which are mapped to the Spartan-MC address space located at 0x1A000 + BASE_ADR + Offset.

Table 7-25: UART registers

Offset	Name	Access	Description
0	UART_STATUS	read/ (write)	Contains the current Rx/Tx FIFO status. For resetting the tx interrupt this register should be write.
1	UART_FIFO_READ	read	Register for incoming data. The LSB of the data word is written to $UART\_FIFO\_READ_0$. For resetting the rx interrupt this register should be read.
2	UART_FIFO_WRITE	write	Register for outgoing data. The LSB of the data word is written $UART\_FIFO\_WRITE_0$.

7.4.2. UART_STATUS Register

Table 7-26: UART status register layout

Bit	Name	Access	Default	Description
0	RX_EMPTY	read	1	Set to one if the Rx FIFO is empty otherwise the value is zero.
1	RX_FULL	read	0	Set to one if the Rx FIFO is full otherwise the value is zero.

Bit	Name	Access	Default	Description
2	TX_EMPTY	read	1	Set to one if the Tx FIFO is empty otherwise the value is zero.
3	TX_FULL	read	0	Set to one if the Tx FIFO is full otherwise the value is zero.
4	TX_IRQ_PRE	read	1	Set to one while data were witten to the Tx FIFO. The value is reset to zero through the next TX access to the Tx FIFO memory.
5	TX_IRQ_FLAG	read	0	Set to one while data were sent. The value is reset to zero through the next write access to TX_IRQ_FLAG.
6-8	x	read	0	Not used.
9	RX_IRQ_ENABLE	read/ write	0/0	To enable rx interrupt set this bit should be set to otherwise set to zero. This bit is only in not polling mode available.
10	TX_IRQ_ENABLE	read/ write	0/0	To enable tx interrupt this bit should be set to one otherwise set to zero. This bit is only in not polling mode available.
11-17	x	read	0	Not used.

Table 7-26: UART status register layout

7.4.3. UART_FIFO_READ Register

Table 7-27: UART status register layout

Bit	Name	Access	Default	Description
0-7	RX	read	0	Register for received data.
8-17	x	read	0	Not used.

7.4.4. UART_FIFO_WRITE Register

Table 7-28: UART status register layout

Bit	Name	Access	Default	Description
0-7	TX	write	0	Register for data to send.
8-17	x	write	x	Not used.

7.4.5. UART C-Header for Register Description

```c
#ifndef UART_LIGHT_H_
#define UART_LIGHT_H_

#ifdef __cplusplus
extern "C" {
#endif

#include <stdint.h>

// Status Signale
#define UART_LIGHT_RX_EMPTY   (1<<0)
#define UART_LIGHT_RX_FULL    (1<<1)
#define UART_LIGHT_TX_EMPTY   (1<<2)
#define UART_LIGHT_TX_FULL    (1<<3)
#define UART_LIGHT_TX_IRQ_PRE (1<<4)
#define   UART_LIGHT_TX_IRQ_FLAG   (1<<5)

// Interruptfreigabe fuer UART light
#define UART_LIGHT_RXIE      (1<<9)
#define UART_LIGHT_TXIE      (1<<10)


typedef struct {
   volatile uint18_t status;      // read/write = Reset Tx
Interrupt
   volatile uint18_t rx_data;   // read = Reset Rx Interrupt
   volatile uint18_t tx_data;   // write
} uart_light_regs_t;

void uart_light_send (uart_light_regs_t *uart, unsigned char
value);
unsigned char uart_light_receive (uart_light_regs_t *uart);
int uart_light_receive_nb (uart_light_regs_t *uart, unsigned
char *value);

#define declare_UART_LIGHT_FILE(uart) { \
   .base_addr = (void*) uart, \
   .send_byte = (fun_stdio_send_byte) uart_light_send, \
   .receive_byte = (fun_stdio_receive_byte) uart_light_receive,
\
   .receive_byte_nb = (fun_stdio_receive_byte_nb)
uart_light_receive_nb \
}
```

```
void __attribute__((error("stdio_uart_light_open
is no longer supported. Declare a global variable
FILE * stdout = &UART_LIGHT_*_FILE instead")))
stdio_uart_light_open(uart_light_regs_t * uart);

#ifdef __cplusplus
}
#endif

#endif /*UART_LIGHT_H_*/
```
Code included from spartanmc/include/peripherals/uart_light.h

8. Serial Peripheral Interface Bus (SPI)

The SPI is a SpartanMC peripheral device for serial communication using the SPI bus. The SPI enables bit frames to be shifted in and out of the component at programmable speed. The frame width can be changed during runtime so that one single SPI master is able to control multiple different slaves. An SPI master can be connected with up to 15 SPI-slaves which share 3 wires:

- SCLK (*serial clock*)

- MOSI (*master out slave in*)

- MISO (*master in slave out*)

Besides the three shared signals above, there is also one dedicated low-active slave-select-signal for each slave. A slave may use the shared wires, only if it has been selected by the master using this select-signal. The block diagram below shows the brief structure of the SPI master and its interfaces to the slave and SpartanMC side. sadf
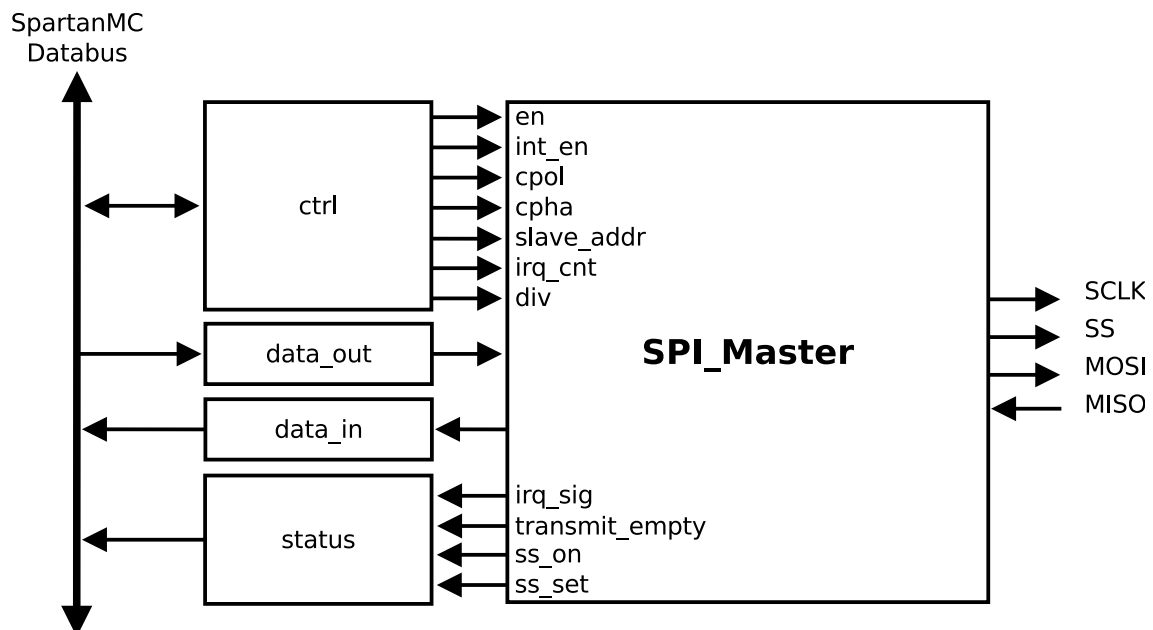
Figure 8-36: SPI block diagram

As shown in this diagram, the SPI master uses four registers on the SpartanMC side, namely `ctrl` , `data_out` , `data_in` and `status` . The former two are writable and the latter two are read-only.The SPI master can be configured by setting the different fields of the `ctrl` register, like the clock divider, the frame width or the slave address etc. After configuration, the data can be sent to the target slave by writing the `data_out` register and the received data can be read from the `data_in` register.

Since an external SPI slave works asynchronously with the SpartanMC, the status of the current transmission should be checked to ensure if it has been finished. This can be done by either polling the `FILL` flag in the `status` register or using the interrupt controller.

8.1. Communication

To start a transfer to a slave, the master has to clear the select signal for this slave. After this, the SPI master can generate the clock signal and shift data to the slave. During each SPI clock cycle, one bit is sent to the slave and one bit is received from the slave. The polarity and phase of the clock can be configured by two bits of the control register, namely CPOL and CPHA, in the following way:

If CPOL = 0, the base value of the clock is zero

- if CPHA = 0, the data are read on the rising edge and refreshed on the falling edge

- if CPHA = 1, the data are read on the falling edge and refreshed on the rising edge

If CPOL = 1, the base value of the clock is one

- if CPHA = 0, the data are read on the falling edge and refreshed on the rising edge

- if CPHA = 1, the data are read on the rising edge and refreshed on the falling edge

In other words, the data are always sampled on the first edge of one clock cycle if CPHA = 0, and on the second one if CPHA = 1, regardless of whether the edge is rising or falling. The timing diagram below shows the clock polarity and clock phase according to an example SPI frame of 8 bits.
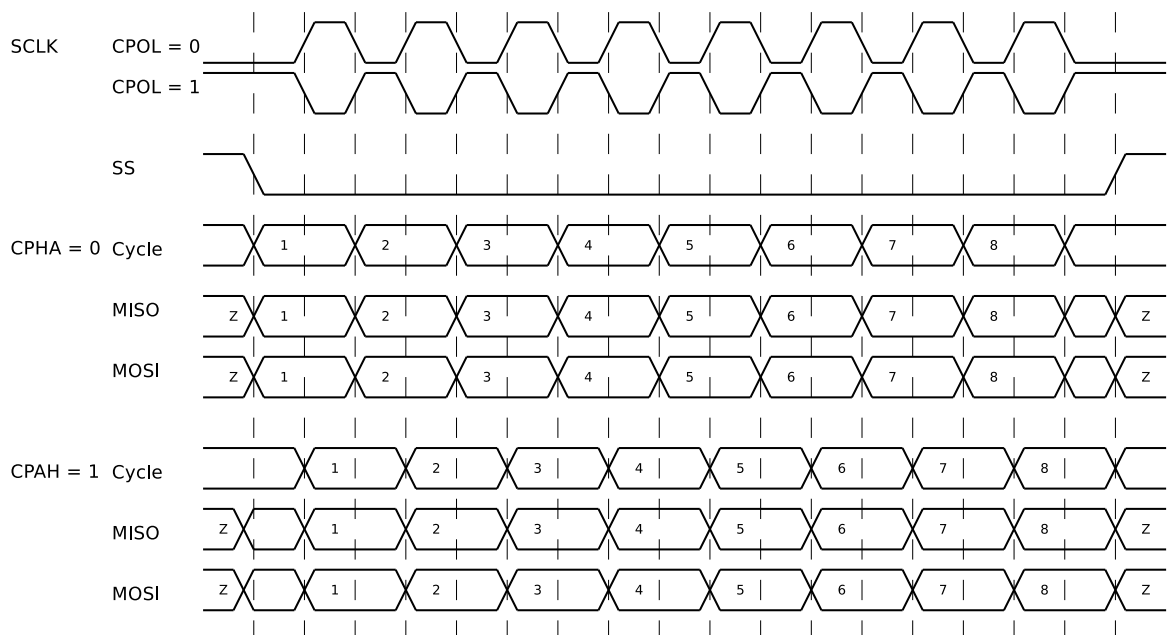
Figure 8-37: SPI frame

Note: By CPHA = 0, the data must be set up one half clock cycle before the first clock edge.

8.2. Module parameter

The SPI master uses only one parameter named `SPI_SS` which gives the number of connected slaves. This parameter can be set in JConfig with respect to the SoC system being built.

Table 8-29: SPI module parameters

Parameter	Default	Descripton
SPI_SS	1	Number of connected slaves

8.3. Peripheral Registers

8.3.1. SPI Register Description

As shown in the table below, the SPI peripheral provides four 18-bit registers which are mapped to the SpartanMC address space.

Table 8-30: SPI registers

Offset	Name	Access	Description
0	spi_control	r/w	Contains the current SPI settings e.g. CPOL, CPHA, clock divider etc.
1	spi_data_out	r/w	Register for outgoing data.
2	spi_data_in	r	Register for incoming data.
3	spi_status	r	Status register.

In the following, the control and status register are described in more details.

8.3.2. SPI Control Register

The table below gives an overview of the layout of the control register:

Table 8-31: SPI control register layout

Bit	Name	Access	Default	Description
0	EN	r/w	0	Enable the master.
1	IRQ_EN	r/w	0	Enable the interrupt.

Bit	Name	Access	Default	Description
2	CPOL	r/w	0	Clock polarity.
3	CPHA	r/w	0	Clock phase.
4-7	SLAVE	r/w	0	Address of the selected slave. 0 deactivates all slave-select-signals 1 activates the first slave-select-signal .... 15 activates the 15th slave-select-signal
8-12	BITCNT	r/w	00000	Number of bits contained in one frame. Only the range from 1 to 18 can be used. Other values will be ignored. 1 : 1-bit SPI frame .... 18 : 18-bit SPI frame
13-15	CLK_DIV	r/w	111	Clock divider. SCLK = (clk_peri) / (4 * divider) $0 : 2^0 = 1$ $1 : 2^1 = 2$ .... $7 : 2^7 = 128$

Table 8-31: SPI control register layout

8.3.3. SPI Status Register

The following table shows the layout of the status register:

Table 8-32: SPI control register layout

Bit	Name	Access	Default	Description
0-2	FILL	r	0	Fill level of the output FIFO: 000: empty 001: 1/4 full 010: 1/2 full 011: 3/4 full 100: full
3	INT	r	0	Is set to 1 together with `TRANS_EMPTY`, but will be cleared on a read access to the `data_in` register.
4	SS_ON	r	0	Is set to 1, if the slave address is not equal 0.

Bit	Name	Access	Default	Description
5	SS_SET	r	1	Is set to 1, if the slave address has been assigned a new value.

Table 8-32: SPI control register layout

8.3.4. SPI C-Header spi.h for Register Description

```c
#ifndef __SPI_H
#define __SPI_H

#ifdef __cplusplus
extern "C" {
#endif

#include <peripherals/spi_master.h>
#include <peripherals/spi_slave.h>
#include <bitmagic.h>

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/spi.h

8.3.5. SPI C-Header spi_master.h for Register Description

```c
#ifndef __SPI_MASTER_H
#define __SPI_MASTER_H

#ifdef __cplusplus
extern "C" {
#endif

#include <peripherals/spi_common.h>

// CONTROL
#define SPI_MASTER_CTRL_EN       SPI_CTRL_EN
#define SPI_MASTER_CTRL_INT_EN   SPI_CTRL_INT_EN
#define SPI_MASTER_CTRL_CPOL     SPI_CTRL_CPOL
#define SPI_MASTER_CTRL_CPHA     SPI_CTRL_CPHA
#define SPI_MASTER_CTRL_SLAVE   0x000F0U // 00 0000 0000 1111 0000
#define SPI_MASTER_CTRL_BITCNT   0x01F00U // 00 0001 1111 0000 0000
#define SPI_MASTER_CTRL_DIV     0x0E000U // 00 1110 0000 0000 0000

//STATUS
#define SPI_MASTER_STAT_FILL      0x00007
```

```
#define SPI_MASTER_STAT_INT      0x00008
#define SPI_MASTER_STAT_SS_ON    0x00010
#define SPI_MASTER_STAT_SS_SET   0x00020

typedef struct {
   spi_t spi;
} spi_master_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/spi_master.h

8.3.6. SPI C-Header spi_slave.h for Register Description

```
#ifndef __SPI_SLAVE_H
#define __SPI_SLAVE_H

#ifdef __cplusplus
extern "C" {
#endif

#include <peripherals/spi_common.h>

#define SPI_SLAVE_CTRL_EN        SPI_CTRL_EN
#define SPI_SLAVE_CTRL_INT_EN    SPI_CTRL_INT_EN
#define SPI_SLAVE_CTRL_CPOL      SPI_CTRL_CPOL
#define SPI_SLAVE_CTRL_CPHA      SPI_CTRL_CPHA
#define SPI_SLAVE_CTRL_DONE      0x00100 // 00 0000 0001 0000
0000
#define SPI_SLAVE_CTRL_INT       0x00200 // 00 0000 0010 0000
0000
#define SPI_SLAVE_CTRL_BITCNT    0x01C00 // 00 0001 1100 0000
0000

typedef struct {
   spi_t spi;
} spi_slave_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```

Code included from spartanmc/include/peripherals/spi_slave.h

8.3.7. SPI C-Header spi_common.h for Register Description

```c
#ifndef __SPI_COMMON_H
#define __SPI_COMMON_H

#ifdef __cplusplus
extern "C" {
#endif

// CONTROL
#define SPI_CTRL_EN       0x00001U // 00 0000 0000 0000 0001
#define SPI_CTRL_INT_EN   0x00002U // 00 0000 0000 0000 0010
#define SPI_CTRL_CPOL     0x00004U // 00 0000 0000 0000 0100
#define SPI_CTRL_CPHA     0x00008U // 00 0000 0000 0000 1000

typedef volatile struct {
   volatile unsigned int spi_control;
   volatile unsigned int spi_data_out;
   volatile unsigned int spi_data_in;
   volatile unsigned int spi_status;
} spi_t;

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/spi_common.h

8.3.8. Basic Usage of the SPI Registers

The structures shown above can be used in a program directly, if `<spi.h>` has been included. They serve as the interface between software and hardware. A programmer can configure and control the SPI master simply using these structures without having to care about any low-level details (e.g. timing) at all. According to several trivial examples, this section illustrates how to use this interface to communicate with the SPI master. First of all, assume that `SPI_MASTER_0` is a pointer of type `spi_master_regs_t` which has been assigned the physical address of a SPI master. The registers of the SPI master can be accessed via this pointer.

- **Example 1 : Enable the SPI master**

  ```c
  SPI_MASTER_0->spi.spi_control |= SPI_MASTER_CTRL_EN;
  ```

- **Example 2 : Set the frame width to 16**

```
/* don't forget to clear the default value */
SPI_MASTER_0->spi.spi_control &= ~SPI_MASTER_CTRL_BITCNT;
SPI_MASTER_0->spi.spi_control |= (16<<8);
```

- **Example 3 : Send the constant value 256 to the slave 1**

```
/* activate the slave 1 */
SPI_MASTER_0->spi.spi_control |= (1 << 4);
/* data written into spi_data_out will be sent */
SPI_MASTER_0->spi.spi_data_out = 256;
/* deactivate the slave 1 */
SPI_MASTER_0->spi.spi_control &= ~SPI_MASTER_CTRL_SLAVE;
```

- **Example 4 : Read the received value**

```
int v = SPI_MASTER_0->spi.spi_data_in;
```

- **Example 5 : Check IRQ_Sig of the status register**

```
/* wait until the bit has been set */
while(!(SPI_MASTER_0->spi.spi_status&SPI_MASTER_STAT_INT));

/* handle the interrupt here */
```

9. I2C Master

I2C (also referred to as *two-wire interface*) is a serial bus which allows for connection of multiple master devices to multiple slave devices, only using two single bidirectional lines:

- SCL (*serial clock line*)

- SDA (*serial data line*)

Both lines need to be pulled up with resistors. Because of this, both of them remain simply high if there is no communication between any master and slave. The clock line needs to be driven by a master. Using this clock, the data will be transmitted bit by bit between the master and the corresponding slave over the data line.

The SpartanMC I2C master controller is a peripheral device which supports I2C functions. It can issue multi-byte transfers without further control from the processor. The following block diagram gives an overview of its structure and interfaces to both the slave and SpartanMC side.
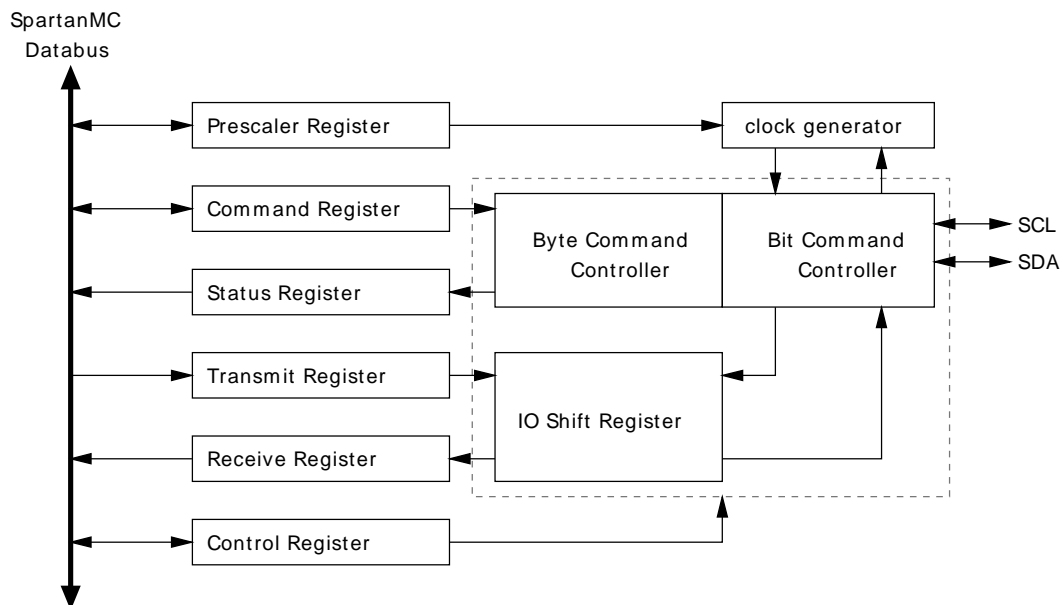
Figure 9-38: I2C block diagram

The I2C master controller can be configured and controlled by setting the writable registers such as the `Control` , `Command` and `Data` registers on the SpartanMC side with flags, commands, slave addresses or data to be sent. With respect to the current settings, it will operate autonomously, i.e. send/receive data to/from slave. After the

current operation, the corresponding bits in the `Status` register will be set and the received data will be written in the `Data` register. The `Status` register is read-only, whereas the `Command` register is write-only.

9.1. Communication

As mentioned above, both SDA and SCL remain high if there is no transmission be-
tween any master and slave. In this case, the I2C bus is considered as idle and can
be used by any master. To start a transmission, SDA is pulled low while SCL remains
high. After the start signal, 8-bit data packets will be transferred, one bit on each rising
edge of SCL. Since multiple slaves can be attached to the I2C bus, each of them should
have a unique 7-bit address so that it can be distinguished from the other slaves. As the
first packet, the master should always put the 7-bit address of the target slave and one
direction bit on the bus. If the direction bit is 1, the master wants to read data from the
slave, otherwise write data to it. After the corresponding slave has received the start
packet, it needs to send 1-bit acknowledge back to the master as response. After this
handshake, the master can begin reading or writing data. If the current transmission is
over, SDA must be released to float high again which is used as stop signal and idle
marker. Except for the start and stop signal, the SDA line only changes while SCL is
low. The timing diagram below shows an example transmission of two data packets.
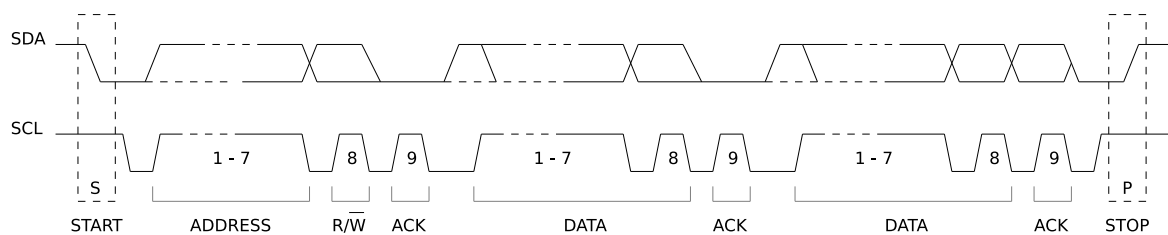
Figure 9-39: SCL, SDA Timing for Data Transmission

Each time after a data packet has been transmitted in one direction, an acknowledge
bit needs to be transmitted in the other direction, as shown in the following diagram.
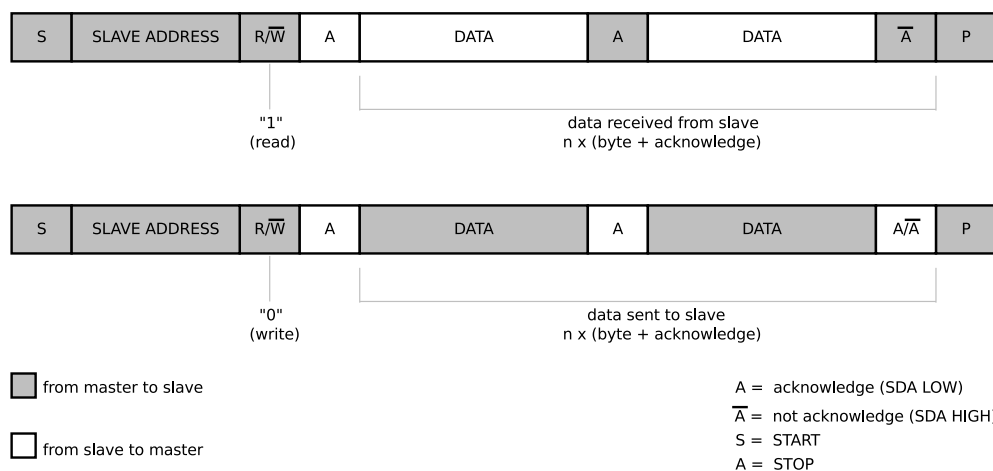
Figure 9-40: I2C Acknowledge

If the transmitter gets a "0" (ACK) as acknowledge, the transmission has succeeded. Otherwise, if it gets a "1", meaning that:

- If the transmitter is master
    - Unknown slave
    - Busy slave
    - Unknown command
- If the transmitter is slave
    - Stop request from the master

9.2. Bus Arbitration

Since multiple masters can be connected to an I2C bus, several of them may start the transmission simultaneously. To overcome this situation, all masters monitor SDA and SCL continuously. If one of them detects that SDA is low while it should actually be high on the next rising edge of SCL, it will stop the current transmission immediately. This process is called *arbitration* and illustrated in the following diagram.
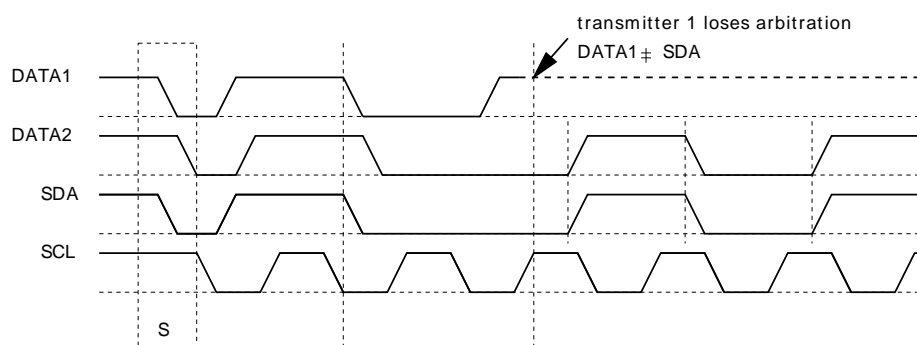
Figure 9-41: I2C Arbitration

9.3. Peripheral Registers

9.3.1. Transfer logic

The peripheral has 8 data registers. They can either contain data to be sent or data that has been received.

When starting a transmission by writing to the `Command` register, you can specify the number of bytes to transfer and the transfer's direction. Regardless of the transfer's direction, the first byte will **always** be written, because read transfers need a slave ad-

dress as first byte. The remaining bytes will be read / written depending on the `Command` register's contents. Bytes received from the slave will be written into the `Data` registers beginning at index 1, so that the slave address remains intact.

If a slave does not acknowledge receipt of a byte, no further bytes will be transmitted in the current transfer. A corresponding flag is then set in the `Status` register.

Interrupts are generated if a transfer has finished, either by a slave not responding, by losing arbitration or by successfully transmitting all bytes. The interrupt flag is reset by any access to any of the peripheral's registers.

9.3.2. I2C Register Description

The I2C peripheral provides eleven 18-bit registers which are mapped to the Spartan-MC address space. In the following, the layout of each register is described in more detail.

Table 9-33: I2C registers

Offset	Name	Access	Description
0-7	DATA	r/w	Contains bytes to be sent or that were received.
8	CONTROL	r/w	Contains a 16-bit clock divider and two enable bits for the I2C master itself and the interrupt controller respectively.
9	COMMAND	w	Used to start transfers.
10	STATUS	r	Contains the controller status flags.

9.3.3. Data Register

The eight data registers are structured identically:

Table 9-34: I2C data register layout

Bit	Name	Access	Default	Description
0-7	data	r/w	undefined	Register for data to be sent / received.
8-17	-	r	0	Not used.

9.3.4. CONTROL Register

Table 9-35: I2C control register layout

Bit	Name	Access	Default	Description
0-15	PRESCALER	r/w	65535	This field is used to set the clock frequency of the SCL line. To change its value the CORE_EN bit must be set to zero. The prescaler factor can be dermined through the following equation: prescaler = (peripheral_clock / (5 * desired_SCL)) -1.
16	EN	r/w	0	Enable I2C core if set to 1.
17	INTR_EN	r/w	0	Enable interrupt output if set to 1.

9.3.5. COMMAND Register

A write to this register with `COUNT` greater than zero triggers a transfer. While a transfer is in progress, writes to this register are ignored.

Table 9-36: I2C command register layout

Bit	Name	Access	Description
0	STA	w	Generate a start condition in this transfer.
1	STO	w	Generate a stop condition in this transfer.
2	RD	w	Read data from the slave. As explained above, the first byte will still be written, only further bytes will be read.
6-3	COUNT	w	Number of bytes to transfer. This includes both bytes that are written and read. For read transfers, this number is therefore one higher than the number of bytes to read. The maximum allowed value is 8.

9.3.6. STATUS Register

Table 9-37: I2C status register layout

Bit	Name	Access	Default	Description
0	TIP	r	0	Is set to 1 when a transfer is in progress.
1	BUSY	r	0	Is set to 1 after a start signal has been detected and set to 0 after a stop signal has occurred.
2	NO_ACK	r	0	Is set to 1 when a slave did not send an acknowledge signal during the transfer. Reset by writing to the `Command` register.
3	AL	r	0	Is set to 1 if the arbitration has been lost.

9.3.7. I2C C-Header i2c_master.h for Register Description

```c
#ifndef __I2C_MASTER_
#define __I2C_MASTER_

#ifdef __cplusplus
extern "C" {
#endif

typedef volatile struct {
    volatile unsigned int data[8];// (r/w)
    volatile unsigned int ctrl;   // (r/w)
    volatile unsigned int cmd;    // (w)
    volatile unsigned int stat;   // (r)
} i2c_master_regs_t;

/* ----- Control register
bits                                      */

#define   I2C_CTRL_EN       (1<<16)   /* Core enable
bit               */
#define   I2C_CTRL_INTR_EN  (1<<17)   /* Interrupt enable
bit         */

/* ----- Command register
bits                                      */
#define I2C_CMD_STA      (1<<0)   /* Generate (repeated) start
condition*/
#define I2C_CMD_STO      (1<<1)   /* Generate stop
condition        */
#define I2C_CMD_RD       (1<<2)   /* Read from
slave              */

/* ----- Status register
bits                                       */

#define I2C_STA_TIP      (1<<0)   /* Transfer in
progress         */
#define I2C_STA_BUSY      (1<<1)   /* Bus is
busy                 */
#define I2C_STA_NO_ACK      (1<<2)   /* Got no ACK (is the
device there?) */
#define I2C_STA_AL       (1<<3)   /* Arbitration
lost          */

#ifdef __cplusplus
```

```
}
#endif

#endif //define __I2C_MASTER
```
Code included from spartanmc/include/peripherals/i2c_master.h

9.3.8. Basic Usage of the I2C Registers

The structure shown above serves as interface between hardware and software. It can be used directly in a C program by including the header file `<i2c_master.h>` . This section presents several quite simple examples to illustrate the usage of this register.

First, assume that `I2C_MASTER_0` is a variable of type `i2c_master_regs_t` .

- **Example 1 : Enable the I2C master and set the prescaler to 134**

   ```
   I2C_MASTER_0.ctrl = I2C_CTRL_EN | 134;
   ```

- **Example 2 : Send write request to the slave at the address 0x42**

   ```
   I2C_MASTER_0.data[0] = 0x42 << 1; // or 0x84
   I2C_MASTER_0.cmd = I2C_CMD_STA | (1 << 3);
   ```

- **Example 3 : Check if the current transfer has finished**

   ```
   /* wait as long as TIP is set */
   while(I2C_MASTER_0.stat & I2C_STA_TIP);

   /* do something here */
   ```

- **Example 4 : Check if a not acknowledge has been received**

   ```
   if(I2C_MASTER_0->stat & I2C_STA_NO_ACK)
       return -1;
   ```

- **Example 5 : Write a constant value 0xFFFF to the slave**

   ```
   I2C_MASTER_0.data[0] = 0xFF;
   I2C_MASTER_0.data[1] = 0xFF;
   I2C_MASTER_0.cmd = 2 << 3;
   ```

- **Example 6 : Read 4 bytes from the slave at address 0x42**

   ```
   I2C_MASTER_0.data[0] = (0x42 << 1) + 1; // or 0x85
   I2C_MASTER_0.cmd = I2C_CMD_RD | I2C_CMD_STA | (5<<3);
   ```

- **Example 7 : Print two received bytes**

   ```
   printf("%x  %x\n",  I2C_MASTER_0.data[1],  I2C_MASTER_0.data
   [2]);
   ```

Note: Sometimes, a hardware manufacturer may give an 8-bit slave ID instead of a 7-bit address. This ID is actually equal `address << 1` and implies that the direction bit is 0. Therefore, it can be sent to the slave as write request directly and `ID + 1` can be used as read request.

10. JTAG-Controller

The JTAG-Controller for the SpartanMC is a JTAG-Master. It can communicate with JTAG-Slaves, which are connected through the 4 JTAG-Pins described in the following Table. If you need a TRST-Port you can use a PortOut Component of the SpartanMC. This component implements an extra feature for MSP430 micro controllers to control the internal clock signal.

Table 10-38: JTAG Basics

Pin	Description
TCK	Test Clock - this pin is the clock signal used for ensuring the timing of the boundary scan system. The TDI shifts values into the appropriate register on the rising edge of TCK. The selected register contents shift out onto TDO on the falling edge of TCK.
TDI	Test Data Input - Test instructions shift into the device through this pin.
TDO	Test Data Output - This pin provides data from the boundary scan registers, i.e. test data shifts out on this pin.
TMS	Test Mode Select - This input which also clocks through on the rising edge of TCK determines the state of the TAP controller.
TRST	This is an optional active low test reset pin. It permits asynchronous TAP controller initialization without affecting other device or system logic.
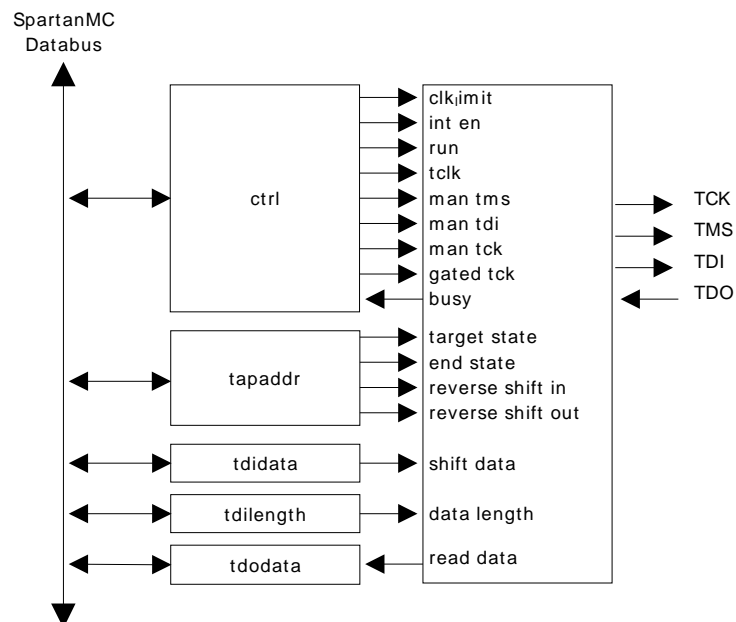
Figure 10-42: JTAG block diagram
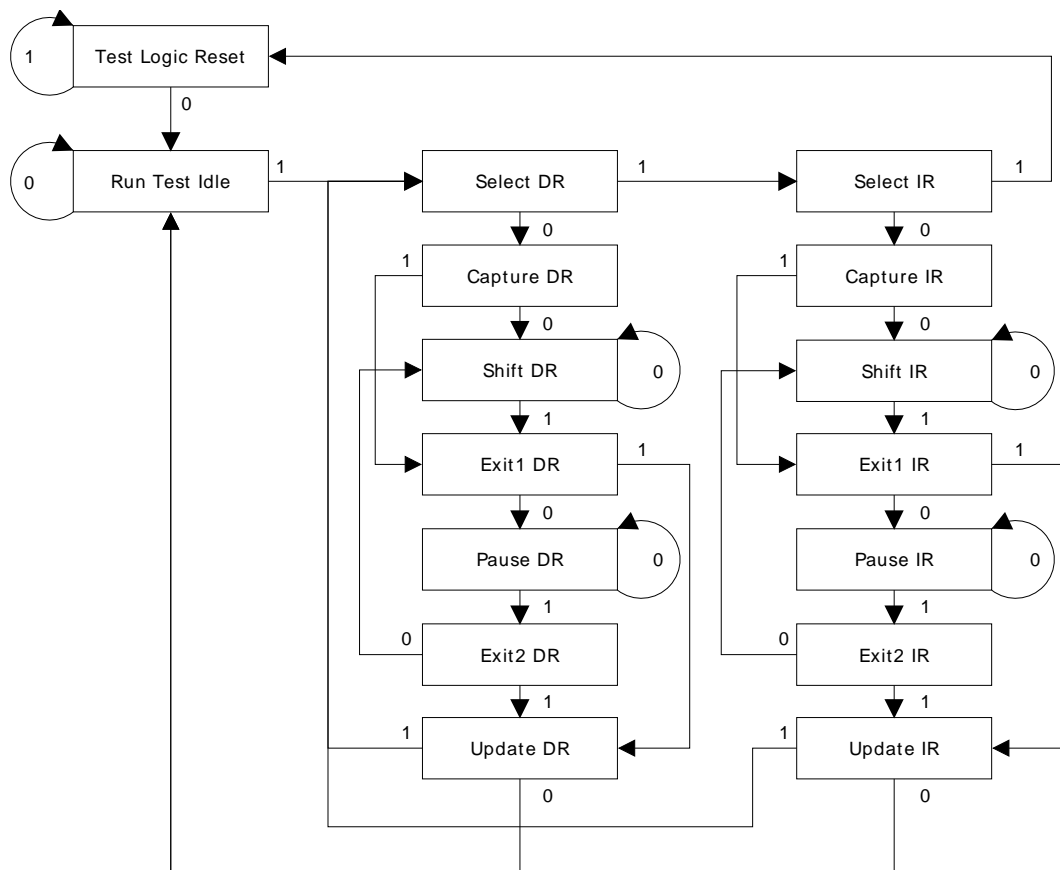
Figure 10-43: JTAG TAP Controller State Machine

10.1. Communication

To shift data to a connected device, you have to bring the TAP Controller in the device into the Run-Test-Idle-State. This can be done by resetting the Controller (clocking TMS=1 6 times) manually and clocking one TMS=0 in to go to the RTI-State. Now the Run-Bit (Bit 11 in the ctrl-register) can be set one and the controller is in automatic mode. Shifting data is very simple. Set target state in the TAP-Control-Register (maybe SHIFT_DR = 4) and set the end state (maybe RUN-TEST-IDLE). Then you put the data into the tdidata register and set the length of this data. Now the controller drives the TAP-Controller in your connected device into the target state (here: 1-0-0), shifts the data and generates a TMS-sequence to drive the TAP-Controller into the end state (here 1-1-0). When the process is done a interrupt is generated when interrupts are enabled. You can also poll the control-register and check the busy-bit (bit 18) to know when the process is done.
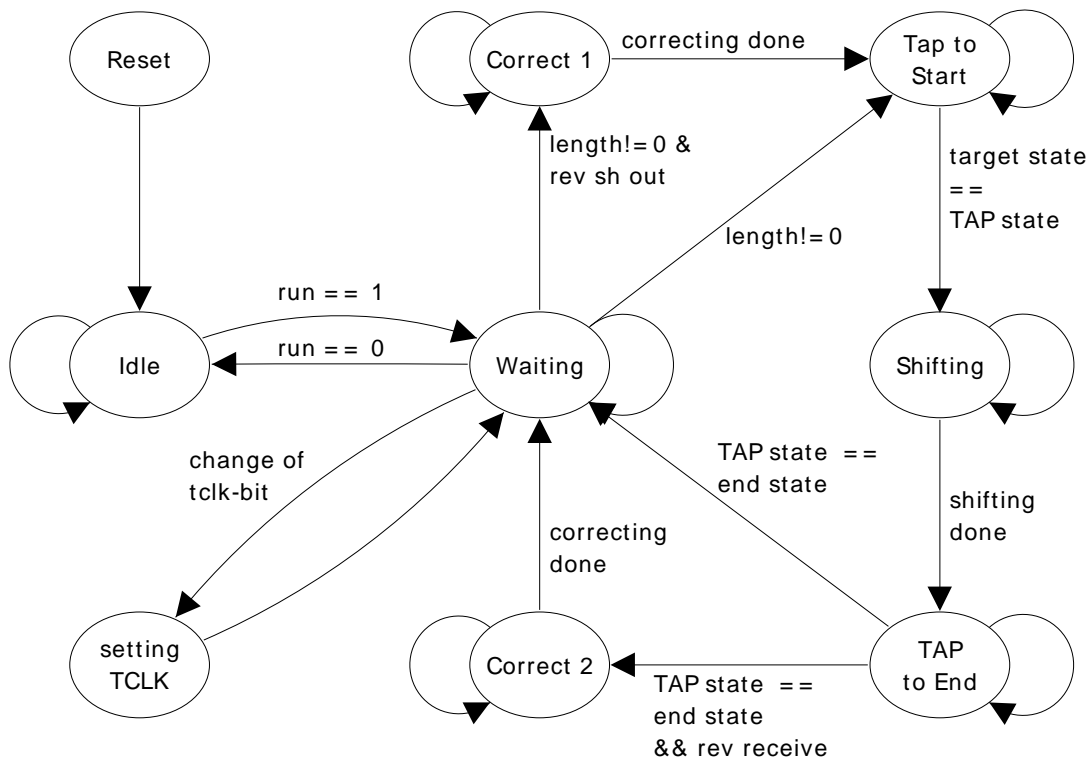
Figure 10-44: JTAG State machine

10.2. Module parameters

This module does not have Synthesis-Parameter

10.3. Peripheral Registers

10.3.1. JTAG Register Description

The JTAG peripheral provides 5 18 bit registers which are mapped to the SpartanMC address space.

Table 10-39: JTAG registers

Offset	Name	Access	Description
0	ctrl	read/ write	Contains the current JTAG setting e.g. clock divider, irq settings, jtag signal levels and generates a busy signal.
1	tapaddresses	read/ write	Register for setting the target- and end state of the shifting process. Also configuring reverse send and reverse receive.
2	tdidata	read/ write	Register for data which should be shifted out.
3	tdilength	read/ write	Register for the length of the data. Writing this registers starts a shift process.
4	tdidata	read	Register for received data.

10.3.2. JTAG Control Register (ctrl)

Table 10-40: JTAG control register layout

Bit	Name	Access	Default	Description
0-9	EN	r/w	1	Clock devive register.
10	IRQ_EN	r/w	0	Enable sending irqs.
11	Run	r/w	0	Enables the automatic mode of the Controller. Before you enable the automatic mode you have to put the TAP controller in your connected device into the RUN-TEST-IDLE mode using the bits for TMS and TCK in this register.
12	TCLK	r/w	0	This bit is a special control bit for Ti-MSP430 microcontroller. It controls the internal clk signal of this microcontroller. Setting TDI to Logic One and holding the TCK for 3 cycles high, set the internal clock signal to one. Setting TDI to zero and holding TCK for three cycles high, set the internal clock signal to zero. The Value of this bit will be set to the clock signal in the MSP430.
13	man TMS	r/w	0	Logical level of the TMS Pin when not in auto mode.

Bit	Name	Access	Default	Description
14	man TDI	r/w	0	Logical level of the TDI Pin when not in auto mode.
15	man TCK	r/w	0	Logical level of the TCK Pin when not in auto mode.
16	gated clk	r/w	0	If this Bit is set to one, the controller does not generate the tck-signal when the controller is idle.
17	busy	r	0	This Bit is set to one when the controller performing a shift operation or changes the tclk-signal (MSP430-feature)

Table 10-40: JTAG control register layout

10.3.3. JTAG TAP Control Register (tapaddr)

Table 10-41: JTAG TAP control register layout

Bit	Name	Access	Default	Description
0-3	TAP target	r/w	0	In this state of the TAP-Controller the provided data will be shifted in.
4-7	TAP end	r/w	0	After shifting the data, the controller will drive the TAP-Controller in the connected device into this state.
8	reverse send	r/w	0	If this bit is set to one, the controller will shift out the MSB of the given data first else the LSB will be shifted out first.
9	reverse receive	r/w	0	If this bit is set to one, the controller will receive the data as MSB else as LSB in the receive shift register.
10-17	not used	-	-	not used

Table 10-41: JTAG TAP control register layout

11. Configurable Parallel Output for 1 to 18 Bit (port_out)

The port output module provides up to 18 output signals. Each output pin can be activated through the corresponding bit in the control register PORT_OUT_OE. If an output is not activated it is set to high-impetance.

11.1. Module Parameters

Table 11-42: PORT_OUT module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
PORT_WIDTH	18	Number of output bits.

11.2. Peripheral Registers

11.2.1. Output Port Register Description

The output port peripheral provides two 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 11-43: PORT_OUT registers

Offset	Name	Access	Description
0	PIN_OUT_DAT	read/ write	Register for outgoing data.
1	PIN_OUT_OE	read/ write	If set to one the corresponding output pin in PIN_OUT_DAT is enabled. After system reset all PIN_BI_OE bits are initialized with zero.

11.2.2. PORT_OUT C-Header for Register Description

```c
#ifndef __PORT_OUT_H
#define __PORT_OUT_H

#ifdef __cplusplus
extern "C" {
#endif

#define PORT_OUTBIT_0 (1<<0)
#define PORT_OUTBIT_1 (1<<1)
#define PORT_OUTBIT_2 (1<<2)
#define PORT_OUTBIT_3 (1<<3)
#define PORT_OUTBIT_4 (1<<4)
#define PORT_OUTBIT_5 (1<<5)
#define PORT_OUTBIT_6 (1<<6)
#define PORT_OUTBIT_7 (1<<7)
#define PORT_OUTBIT_8 (1<<8)
#define PORT_OUTBIT_9 (1<<9)
#define PORT_OUTBIT_10 (1<<10)
#define PORT_OUTBIT_11 (1<<11)
#define PORT_OUTBIT_12 (1<<12)
#define PORT_OUTBIT_13 (1<<13)
#define PORT_OUTBIT_14 (1<<14)
#define PORT_OUTBIT_15 (1<<15)
#define PORT_OUTBIT_16 (1<<16)
#define PORT_OUTBIT_17 (1<<17)

typedef struct port_out {
   volatile unsigned int data; // (r/w)
   volatile unsigned int oe; // (r/w)
} port_out_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/port_out.h

12. Configurable Parallel Input for 1 to 18 Bit (port_in)

The input port module provides up to 18 input signals. These inputs can be used for switches, buttons etc.. Furthermore, they can be configured to generate interrupts (triggerd by a raising or falling edge) as SoC input. Each input pin provides seperate configuration bits.

12.1. Module Parameters

Table 12-44: PORT_IN module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
PORT_WIDTH	18	Number of input bits.

12.2. Interrupts

An interrupt signals will be generated for each enabled PORT_IN bit. To delete the interrupt flag a read access on PIN_IN_DAT, PIN_IN_IE or PIN_IN_EDGSEL is required.

12.3. Peripheral Registers

12.3.1. Input Port Register Description

The input port peripheral provides four 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 12-45: PORT_IN registers

Offset	Name	Access	Description
0	PIN_IN_DAT	read	Register for incomming data.

Offset	Name	Access	Description
1	PIN_IN_IE	read/ write	Enables the interrupts on PIN_IN_DAT register. After system reset all PIN_IN_IE bits are initialized with zero.
2	PIN_IN_EDGSEL	read/ write	Specify the input edge which triggers the interrupt (0 = falling edge, 1 = raising edge) After system reset all PIN_IN_EDGSEL bits are initialized with zero.
3	PIN_IN_IR_STATUS	read	Register for interrupt flags. If set to one it indicates an interrupt on the corresponding input pin. The interrupt flag will be deleted with a read access on all other module registers except this one. After system reset all PIN_IN_IR_STATUS bits are initialized with zero.

12.3.2. PORT_IN C-Header for Register Description

```c
#ifndef __PORT_IN_H
#define __PORT_IN_H

#ifdef __cplusplus
extern "C" {
#endif

#ifndef __PORT_IN_F_H

#define PORT_INBIT_0   (1<<0)
#define PORT_INBIT_1   (1<<1)
#define PORT_INBIT_2   (1<<2)
#define   PORT_INBIT_3   (1<<3)
#define PORT_INBIT_4   (1<<4)
#define PORT_INBIT_5   (1<<5)
#define PORT_INBIT_6   (1<<6)
#define PORT_INBIT_7   (1<<7)
#define PORT_INBIT_8   (1<<8)
#define PORT_INBIT_9   (1<<9)
#define PORT_INBIT_10   (1<<10)
#define PORT_INBIT_11   (1<<11)
#define PORT_INBIT_12   (1<<12)
#define PORT_INBIT_13   (1<<13)
#define PORT_INBIT_14   (1<<14)
#define PORT_INBIT_15   (1<<15)
#define PORT_INBIT_16   (1<<16)
#define PORT_INBIT_17   (1<<17)

#endif
```

```
typedef struct port_in {
   volatile unsigned int data;    // (r)   (r = reset-interrupt)
   volatile unsigned int ie;    // (r/w) (r = reset-interrupt)
   volatile unsigned int edgsel;    // (r/w) (r = reset-
interrupt)
   volatile unsigned int ir_stat;    // (r)
} port_in_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```

Code included from spartanmc/include/peripherals/port_in.h

13. Parallel Input/Output for 1 to 18 Bit (port_bi)

The bidirectional port module provides up to 18 inputs or outputs. Each signal pin can be configured through the corresponding bit in the control registers (PIN_BI_DIR, PIN_BI_OE). If configured as input they can be used to generate interrupts (triggerd by a raising or falling edge) on the SoC inputs. If configured as output the pins can be drove in open drain mode or as tri-state output. (The usage in open drain mode requires at least one pull up resistor.)

13.1. Module Parameters

Table 13-46: Bidirectional port module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
PORT_WIDTH	18	Number of Input/Output Bits.
OD_OUTPUT	0	Specify the output mode (0 = tri-state output, 1 = open drain output).

13.2. Interrupts

An interrupt signals will be generated for each enabled PORT_BI bit. To delete the interrupt flag a read access on PIN_BI_DAT, PIN_BI_IR_EDGSEL, PIN_BI_IE, PIN_BI_DIR or PIN_BI_OE is required.

13.3. Peripheral Registers

13.3.1. PORT_BI Register Description

The bidirectional port peripheral provides six 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 13-47: PORT_BI registers

Offset	Name	Access	Description
0	PIN_BI_DAT	read/ write	Register for incomming or outgoing data.
1	PIN_BI_IE	read/ write	Enables the interrupts on PIN_BI_DAT register. After system reset all PIN_BI_IE bits are initialized with zero.
2	PIN_BI_OE	read/ write	If set to one the corresponding output pin in PIN_BI_DAT is enabled. After system reset all PIN_BI_OE bits are initialized with zero.
3	PIN_BI_DIR	read/ write	Specify the direction (input/output) of the port signal. After system reset all PIN_BI_DIR bits are initialized with zero.
4	PIN_BI_EDGSEL	read/ write	Specify the input edge which triggers the interrupt (0 = falling edge, 1 = raising edge) After system reset all PIN_BI_EDGSEL bits are initialized with zero.
5	PIN_BI_IR_STATUS	read	Register for interrupt flags. If set to one it indicates an interrupt on the corresponding input pin. The interrupt flag will be deleted with a read access on all other module registers except this one. After system reset all PIN_BI_IR_STATUS bits are initialized with zero.

13.3.2. PORT_BI C-Header for Register Description

```c
#ifndef __PORT_BI_H
#define __PORT_BI_H

#ifdef __cplusplus
extern "C" {
#endif

#define PORT_IOBIT_0   (1<<0)
#define PORT_IOBIT_1   (1<<1)
#define PORT_IOBIT_2   (1<<2)
#define PORT_IOBIT_3   (1<<3)
#define PORT_IOBIT_4   (1<<4)
#define PORT_IOBIT_5   (1<<5)
#define PORT_IOBIT_6   (1<<6)
#define PORT_IOBIT_7   (1<<7)
#define PORT_IOBIT_8   (1<<8)
#define PORT_IOBIT_9   (1<<9)
#define PORT_IOBIT_10 (1<<10)
#define PORT_IOBIT_11 (1<<11)
#define PORT_IOBIT_12 (1<<12)
#define PORT_IOBIT_13 (1<<13)
#define PORT_IOBIT_14 (1<<14)
#define PORT_IOBIT_15 (1<<15)
#define PORT_IOBIT_16 (1<<16)
#define PORT_IOBIT_17 (1<<17)

typedef struct port_bi {
   volatile unsigned int data; // (r/w) (reset-interrupt)
   volatile unsigned int ie; // (r/w) (reset-interrupt)
   volatile unsigned int oe; // (r/w)
   volatile unsigned int dir; // (r/w) (reset-interrupt) 1 =
Input
   volatile unsigned int edgsel; // (r/w) (reset-interrupt) 1 =
positiv
   volatile unsigned int ir_stat; // (r)
} port_bi_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/port_bi.h

14. SpartanMC Core Hardware Debugging Support

The SpartanMC Core contains optional Hardware Debugging Support, that allows hardware breakpoints, watchpoints and stepping. Hardware Debugging is enabled on a per-Core basis with the **HARDWARE_DEBUGGING** flag. The number of available break-/watchpoints can be configured from JConfig, as well as the trap indexes that will be called upon registering a hit on either breakpoint, watchpoint or step event.

Breakpoints are placed on single instruction addresses

Watchpoints are placed on Word-Level Memory addresses (each address maps to an 18-bit word). Their configuration allows limiting a watchpoint to lower/upper byte (9-bit) or watching only read or write access.
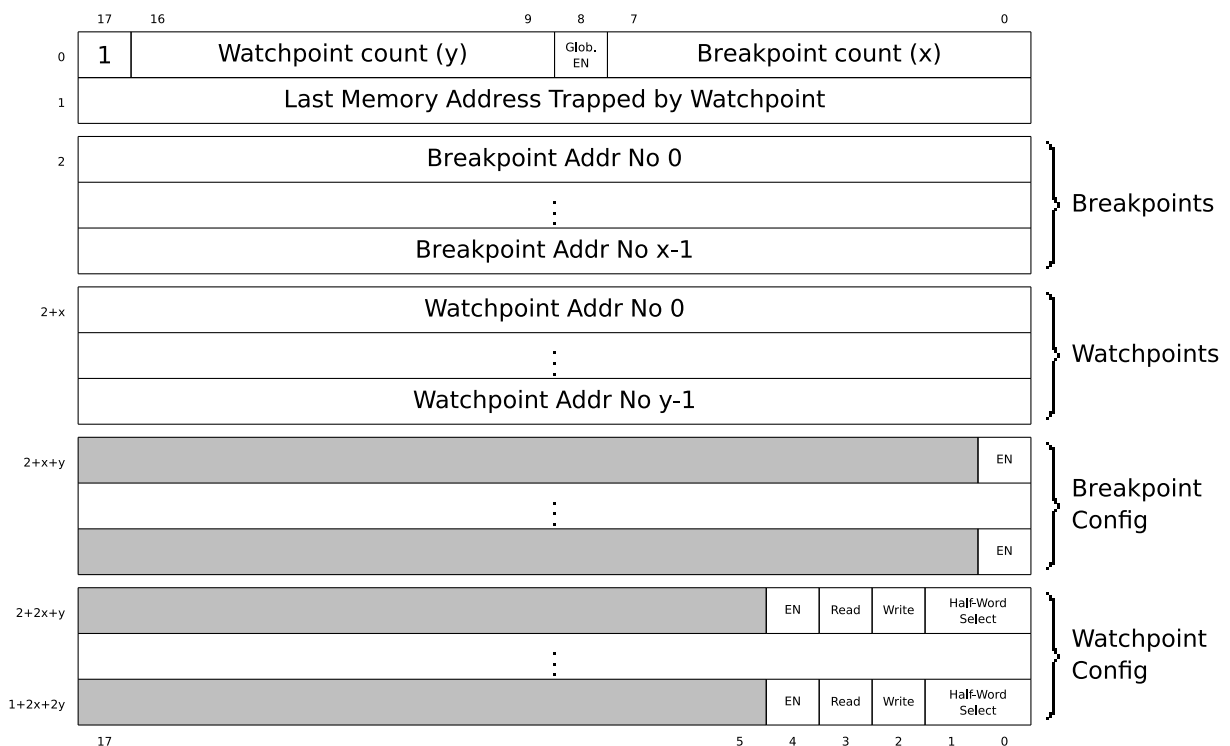
Figure 14-45: Hardware Debugging Registers

14.1. Access

All registers are accessed indirectly by using SFR_DBG_IDX as index, and SFR_DBG_DAT as data.

14.2. Hardware Debugging Status Register (idx 0)

The first register allows detection of the presence of Hardware debugging support (bit 18 is always 1 if present). It also contains the number of break-/watchpoints that were synthesized.

Only writing the following values to the first register has a function

0x00000: turn of Hardware Debugging Functionality globally

0x00001 or 0x00100: turn on Hardware Debugging Functionality globally

0x00002 or 0x00003: Execute a single step upon leaving the current register window
0x00003 combines global enable with single step

14.3. Hardware Behavior

The main work is done by the "Breakpoint Manager". After a reset it will always be disabled. If it is enabled by writing to the**Hardware Debugging Status Register**, it will internally delay its activation, until the register window has changed. This is so that it can never trap a breakpoint in the breakpoint handler code itself, but consequently means, it *must* be activated from its own function. Additionally, breakpoints on the first instruction after activation, may be ignored. Either, if the manager is in single stepping mode (to force it execute at least 1 instuction) or if the address is that of the last breakpoint. The last breakpoint is initialized to 0.

14.4. Last Trap Register

Deprectated: *This register is still present internally, but not curently exposed via Specialfunction Registers.* This information is convenient to gauge, which breakpoint actually hit, but is irrelevant to the debugger, because if it differs from the actual PC, we will have already missed the breakpoint. Software should not place breakpoints on delay slots.

Contains the exact PC of the instruction that caused a trap last. Read-Only. Accounts for delayed jumps/branches. Accounts for pipeline for watchpoints: It will point to instruction causing the memory access, even though several other instructions have since been queued into the pipeline.

14.4.1. Last Trapped Memory Adress Register (idx 1)

Contains the lower 18 bits of the memory address that caused the latest watchpoint. Read-Only.

15. Basic Timer (Timer)

The Basic Timer module can be used to devide the system clock frequency to a user defined periodic signal required by the application. For this purpose the Basic Timer provides a configurable prescaler. If enabled, the prescaler allows the usage of all powers of two between 2 and 256 as prescaler value. The input of prescaler block could be connected to the system clock, a dedicated DCM output or to the output (data register) of a previous timer module. The output of the prescaler is used as input of an 18 bit counter which counts up to a programable value and restarts afterwards.
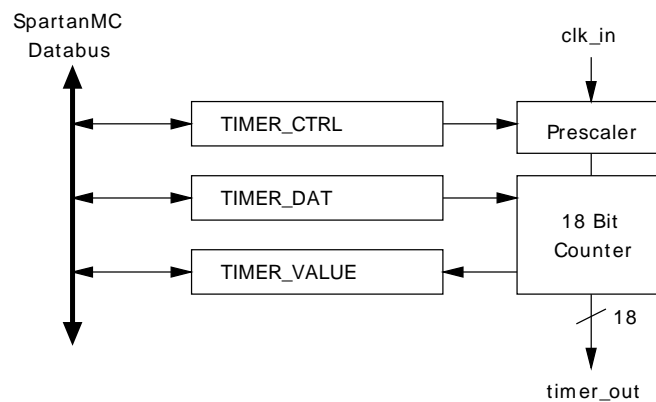
Figure 15-46: Timer block diagram

The basic timer could be used for the following peripherals: Real Time Interrupt (RTI), Watchdog, Capture-, Compare-Module and Pulse-Accumulator-Module. The capture- and compare-module require on their input a complete 18 bit counter register output. While all other modules (RTI, Pulse-Accumulator, Watchdog and additional basic timer) require only one single output bit for their clock input.

15.1. Module parameters

Table 15-48: TIMER module parameters

Parameter	Default Value	Descripton
BASE_ADR	0x10	Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**

15.2. Peripheral Registers

15.2.1. Timer Register Description

The timer peripheral provides three 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 15-49: TIMER registers

Offset	Name	Access	Description
0	TIMER_CTRL	read/ write	Configuration of the timer.
1	TIMER_DAT	read/ write	Maximum value of the 18 bit counter.
2	TIMER_VALUE	read/ write	Current counter value.

15.2.2. TIMER_CTRL Register

Table 15-50: TIMER_CTRL register layout

Bit	Name	Access	Default	Description
0	TI_EN	read/ write	0	If set to one the timer is enabled.
1	TI_PRE_EN	read/ write	0	If set to one the prescaler is enabled.
2-4	TI_PRE_VAL	read/ write	000	Sets the prescaler value : $000 = 2^1$ $001 = 2^2$ $010 = 2^3$ $011 = 2^4$ $100 = 2^5$ $101 = 2^6$ $110 = 2^7$ $111 = 2^8$
5-17	x	read	0	Not used.

Table 15-50: TIMER_CTRL register layout

15.2.3. TIMER_DAT Register

Table 15-51: TIMER_DAT register layout

Bit	Name	Access	Default	Description
0-17	Max Counter	read/write	x	Register for the maximum counter value.

15.2.4. TIMER_VALUE Register

Table 15-52: TIMER_VALUE register layout

Bit	Name	Access	Default	Description
0-17	Main Counter	read/write	0	Register for the current counter value. The content of this 18 bit register is used as timer_output and could be connected to other peripherals e.g. capture- or compare-logic. A single bit of this register could also be used to cascade multiple timers.

15.2.5. TIMER C-Header for Register Description

```c
#ifndef __TIMER_H
#define __TIMER_H

#ifdef __cplusplus
extern "C" {
#endif

#define TIMER_EN       (1<<0)
#define TIMER_PRE_EN   (1<<1)
#define TIMER_PRE_VAL  (1<<2)      // *0 fuer 2^1 bis *7 fuer
2^8
#define TIMER_PRE_2   (TIMER_PRE_VAL * 0)
#define TIMER_PRE_4   (TIMER_PRE_VAL * 1)
#define TIMER_PRE_8   (TIMER_PRE_VAL * 2)
#define TIMER_PRE_16   (TIMER_PRE_VAL * 3)
#define TIMER_PRE_32   (TIMER_PRE_VAL * 4)
#define TIMER_PRE_64   (TIMER_PRE_VAL * 5)
#define TIMER_PRE_128 (TIMER_PRE_VAL * 6)
#define TIMER_PRE_256 (TIMER_PRE_VAL * 7)

typedef struct timer {
```

```
    volatile unsigned int control;
    volatile unsigned int limit;
    volatile unsigned int value;
} timer_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/timer.h

16. Timer Capture Module (timer-cap)

The timer capture module is used to capture the value of a timer register after an external trigger signal.

Note: **The timer capture module always requires a basic timer module as input. Hence, it can not work autonomously.**

(Otherwise, a basic timer could be used as input for multiple capture moduls.)
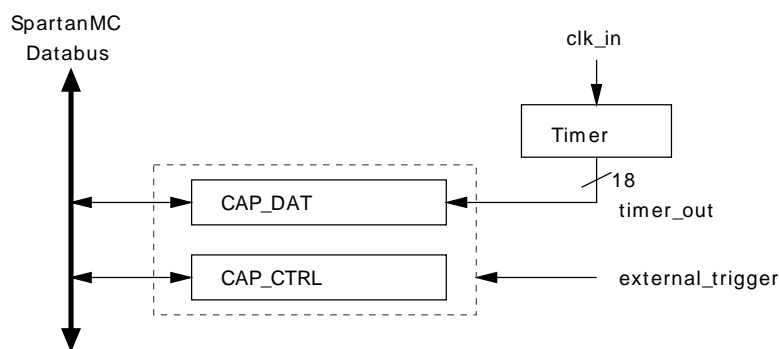
Figure 16-47: Capture module block diagram

16.1. Usage and Interrupts

If this module is triggerd by an external Signal, the current timer value is stored in the local capture register.

Optionally, an interrupt could be generated for each capture event. The interrupt flag is cleared with an access on the data register or control register.

The capture module provides two operation modes for interrupt generation: On the one hand, it could generate its interrupt on edges of a input signal. On the other hand, it could generate its interrupt during a specific level of the input signal. After completion of the capture procedure the enable bit in the control register is cleared. To start a new capture procedure this bit has to be set again.

16.2. Module parameters

Table 16-53: TIMER Capture module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**

16.3. Peripheral Registers

16.3.1. Timer Capture Register Description

The timer capture module provides two 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 16-54: Timer capture registers

Offset	Name	Access	Description
0	CAP_CTRL	read/ write	Configuration of the operation mode. (An acces on this register clears the interrupt flag)
1	CAP_DAT	read	Register for captured data. (An acces on this register clears the interrupt flag)

16.3.2. CAP_DAT Register

Table 16-55: CAP_DAT register layout

Bit	Name	Access	Default	Description
0-17	Capture Value	read	x	The captured data.

16.3.3. CAP_CTRL Register

Table 16-56: CAP_CTRL register layout

Bit	Name	Access	Default	Description
0	CAP_EN	read/write	0	If set to one the capture logic is enabled. This bit is cleared after capture event completion.
1	CAP_EN_INT	read/write	0	If set to one the interrupt is enabled.
2-4	CAP_MODE	read/write	000	Sets the operation mode: 000 = capture disable 001 = not used 010 = capture on falling edge 011 = capture on raising edge 100 = capture on low input signal level 101 = capture on high input signal level 110 = capture on all edges 111 = capture on all edges
5-17	x	read	0	Not used.

Table 16-56: CAP_CTRL register layout

16.3.4. TIMER_CAP C-Header for Register Description

```c
#ifndef __CAPTURE_H
#define __CAPTURE_H

#ifdef __cplusplus
extern "C" {
#endif

#define CAPTURE_EN              (1 << 0)
#define CAPTURE_EN_INT          (1 << 1)
#define CAPTURE_EDGE            (1 << 2)

#define CAPTURE_NON             (CAP_EDGE * 0)
#define CAPTURE_FALLING_EDGE    (CAP_EDGE * 2)
#define CAPTURE_RISING_EDGE      (CAP_EDGE * 3)
#define CAPTURE_LOW_LEVEL       (CAP_EDGE * 4)
#define CAPTURE_HIGH_LEVEL       (CAP_EDGE * 5)
#define CAPTURE_ANYTHING_EDGE   (CAP_EDGE * 7)

typedef struct cap {
   volatile unsigned int   CAP_CTRL;   // (r/w)
   volatile unsigned int   CAP_DAT;   // (r)
} capture_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/capture.h

17. Timer Compare Module (timer-cmp)

The timer compare module is used to generate variable frequencies or programmable duty cycles by comparing an internal value to a given timer value.

Note: **The timer compare module always requires a basic timer module as input. Hence, it can not work autonomously.**

(Otherwise, a basic timer could be used as input for multiple capture moduls.)
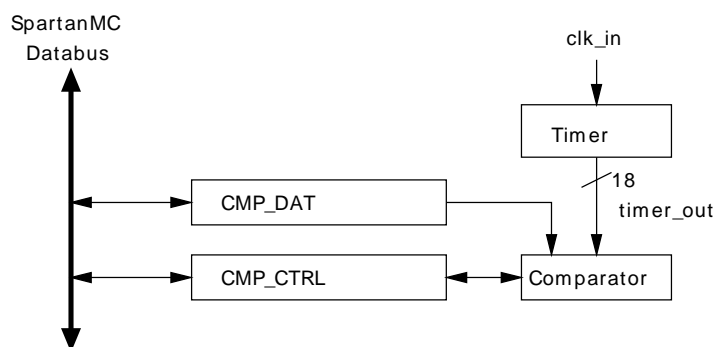
Figure 17-48: Timer compare module block diagram

17.1. Usage and Interrupts

If the programmend value of the compare register equals the current timer value the timer compare module triggers an event. These events could be the generation of an interrupt or the switching of the output pin (set, reset, or negate). In case of an interrupt generation, the interrupt is cleared on each access to the modules registers. In case the module output pin is used, the compare module contains a control register which specifies the behavior of this pin.

17.2. Module parameters

Table 17-57: TIMER Compare module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**

17.3. Peripheral Registers

17.3.1. Timer Compare Register Description

The timer compare module provides tow 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 17-58: Timer Compare registers

Offset	Name	Access	Description
0	CMP_CTRL	read/ write	Specify the operation mode. (An access on this register clears the interrupt flag)
1	CMP_DAT	read/ write	Compare value for the 18 bit counter of the basic timer modules.

17.3.2. Compare Control Register

Table 17-59: CMP_CTRL register layout

Bit	Name	Access	Default	Description
0	CMP_EN	read/ write	0	If set to one the compare logic is enabled.
1	CMP_EN_INT	read/ write	0	If set to one the interrupt is enabled.
2-4	CMP_MODE	read/ write	000	Operation mode (if bit 4 = 0): 000 = Output remains constant 001 = Set output (After trigger event the oputput is always set to 1). 010 = Clear output (After trigger event the oputput is always set to 0). 011 = Toggle output after trigger event
4	OUT_TYP	read/ write	0	If the fourth bit of the operation mod register is set to 1 the output pin switches two times per period. Firstly, on each zero crossing and secondly on the configured maximum value (COMP_DAT). This mechanism enables the usage of the compare module for pulse width modulation (PWM).
2-4	CMP_MODE	read/ write	000	Opertaion mode (if bit 4 = 1): 100 = Output remains constant 101 = Output is set to 1 if timer value equals COMP_DAT -- output is set to 0 if timer value equals 0. 110 = Output is set to 0 if timer value equals COMP_DAT -- output is set to 1 if timer value equals 0.

Bit	Name	Access	Default	Description
				111 = Output is set to 1 if timer value equals COMP_DAT -- output is set to 0 if timer value equals 0.
5	CMP_EN_OUT	read/ write	0	If set to one the comparator output is enabled.
6	CMP_VAL_OUT	read	0	Comparator output bit.
7-17	x	read	0	Not used.

Table 17-59: CMP_CTRL register layout

17.3.3. Compare Value Register

Table 17-60: CMP_DAT register layout

Bit	Name	Access	Default	Description
0-17	CMP_DAT	read/ write	x	18 bit compare value

17.3.4. TIMER_CMP C-Header for Register Description

```
#ifndef __COMPARE_H
#define __COMPARE_H

#ifdef __cplusplus
extern "C" {
#endif

#define COMPARE_EN        (1 << 0)   // Compare Enable
#define COMPARE_EN_INT    (1 << 1)   // Compare Interrupt Enable
#define COMPARE_MODE      (1 << 2)   // Mode Bit 0

#define COMPARE_NON_FRQ   (COMPARE_MODE * 0) // Ausgang bleibt
gleich
#define COMPARE_SET_OUT   (COMPARE_MODE * 1) // Ausgang
setzen(=1)
#define COMPARE_CLEAR_OUT (COMPARE_MODE * 2) // Ausgang
zurücksetzen(=0)
#define COMPARE_TOGGLE_OUT (COMPARE_MODE * 3) // Ausgang
negieren
#define COMPARE_NON_IMP   (COMPARE_MODE * 4) // Ausgang bleibt
gleich
#define COMPARE_C0_N1    (COMPARE_MODE * 6) // Ausgang auf 0
wenn Timer = CMP_DAT ist -- Ausgang auf 1 wenn Timer = 0 ist.
```

```
#define COMPARE_C1_N0   (COMPARE_MODE * 7) // Ausgang auf 1
wenn Timer = CMP_DAT ist -- Ausgang auf 0 wenn Timer = 0 ist.

#define COMPARE_EN_OUT   (1 << 5)   // Compare Output Enable
#define COMPARE_VAL_OUT   (1 << 6)    // Compare Output Value

typedef struct cmp {
   volatile unsigned int CMP_CTRL; // (r/w)
   volatile unsigned int CMP_DAT; // (r/w)
} compare_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/compare.h

18. Timer Real Time Interrupt Module (timer-rti)

The Timer RTI module can be used to divide the system clock frequency to a user defined periodic signal required by the application. For this purpose the Timer RTI provides a configurable prescaler. If enabled, the prescaler allows the usage of all powers of two between 2 and 32768 as prescaler value. The input of the prescaler block can be connected to the system clock, a dedicated DCM output or to the output of a previous timer module. The output of the prescaler could be connected to another timer module or could be used to generates an interrupt which for the application.
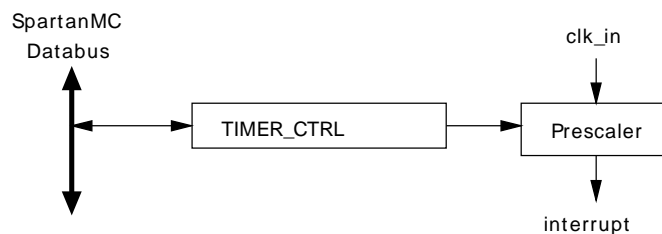
Figure 18-49: Timer RTI block diagram

Note: The timer RTI module could be used as stand alone peripheral or in connection with another timer module (used as input).

18.1. Interrupts

The peripheral generates a cyclic interrupt signals on the maximum value of the timer period.

18.2. Module Parameters

Table 18-61: Timer RTI module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**

18.3. Peripheral Registers

18.3.1. Timer RTI Register Description

The timer RTI peripheral provides one 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 18-62: TIMER RTI registers

Offset	Name	Access	Description
0	RTI_CTRL	read/ write	Specify the operation mode. (An access on this register clears the counter value)

18.3.2. RTI_CTRL Register

Table 18-63: RTI_CTRL register layout

Bit	Name	Access	Default	Description
0	RTI_EN	read/ write	0	If set to one the timer RTI logic is enabled.
1	RTI_EN_INT	read/ write	0	If set to one the timer RTI interrupt is enabled.
2-5	RTI_PRE_VAL	read/ write	0000	Specify the prescaler value: 0000 = 2^0 0001 = 2^1 0010 = 2^2 0011 = 2^3 0100 = 2^4 0101 = 2^5 0110 = 2^6 0111 = 2^7 1000 = 2^8 1001 = 2^9 1010 = 2^10 1011 = 2^11 1100 = 2^12 1101 = 2^13 1110 = 2^14

Bit	Name	Access	Default	Description
				1111 = 2^15
6-17	x	read	0	Not used.

Table 18-63: RTI_CTRL register layout

18.3.3. RTI C-Header for Register Description

```
#ifndef __RTI_H
#define __RTI_H

#ifdef __cplusplus
extern "C" {
#endif

#define RTI_EN         (1 << 0)
#define RTI_EN_INT      (1 << 1)
#define RTI_PRE_VAL      (1 << 2) // *0 fuer 2^0 bis *15 fuer
2^15

#define RTI_PRE_1      (RTI_PRE_VAL * 0)
#define RTI_PRE_2      (RTI_PRE_VAL * 1)
#define RTI_PRE_4      (RTI_PRE_VAL * 2)
#define RTI_PRE_8      (RTI_PRE_VAL * 3)
#define RTI_PRE_16      (RTI_PRE_VAL * 4)
#define RTI_PRE_32      (RTI_PRE_VAL * 5)
#define RTI_PRE_64      (RTI_PRE_VAL * 6)
#define RTI_PRE_128      (RTI_PRE_VAL * 7)
#define RTI_PRE_256      (RTI_PRE_VAL * 8)
#define RTI_PRE_512      (RTI_PRE_VAL * 9)
#define RTI_PRE_1024   (RTI_PRE_VAL * 10)
#define RTI_PRE_2048   (RTI_PRE_VAL * 11)
#define RTI_PRE_4096   (RTI_PRE_VAL * 12)
#define RTI_PRE_8192   (RTI_PRE_VAL * 13)
#define RTI_PRE_16384   (RTI_PRE_VAL * 14)
#define RTI_PRE_32765   (RTI_PRE_VAL * 15)

typedef struct rti {
   volatile unsigned int ctrl;
} rti_regs_t;

#ifdef __cplusplus
}
#endif
```

```
#endif
```
Code included from spartanmc/include/peripherals/rti.h

19. Timer Pulse Accumulator Module (timer-pulseacc)

The timer pulse accumulator module counts impulses from an external input. The module supports two operation modes: Either it counts impulses on an input called PIN or it counts impulses on RTI input until the next impulse on PIN.
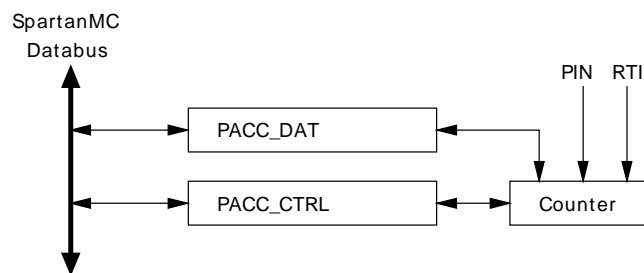
Figure 19-50: Timer Pulse Accumulator block diagram

In the first operation mode the module counts continously all impulses from the input PIN. In the second mode the counter stops if an impulse on RTI occurs. If the counter has stopped (due to an RTI impulse) a read access to the counter register will clear the counter value. Whereas a read access to the control register always clears the counter value in both operation modes.

Note: The timer pulse accumulator can be used as stand alone peripheral or in connection with an basic timer module (used as impulse source).

19.1. Module Parameters

Table 19-64: Timer Pulse Accumulator module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**

19.2. Peripheral Registers

19.2.1. Timer Pulse Accumulator Register Description

The timer pulse accumulator peripheral provides two 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 19-65: Timer Pulse Accumulator Registers

Offset	Name	Access	Description
0	PACC_CTRL	read/ write	Specify the operation mode. (An access on this register clears the counter value)
1	PACC_DAT	read	Counter value register.

19.2.2. PACC_CTRL Register

Table 19-66: PACC_CTRL register layout

Bit	Name	Access	Default	Description
0	PACC_EN	read/ write	0	If set to one the pulse accumulator logic is enabled.
1	PACC_MODE	read/ write	0	Operation mode: 0 = Count all impulses (raising edges) on input PIN. 1 = Count all impulses (raising edges) on imput RTI until an input on PIN occurs.
2-17	x	read	0	Not used.

Table 19-66: PACC_CTRL register layout

19.2.3. PACC_DAT Register

Table 19-67: PACC Counter register layout

Bit	Name	Access	Default	Description
0-17	Counter	read/ write	x	18 bit counter value.

19.2.4. PACC C-Header for Register Description

```c
#ifndef __COUNTER_H
#define __COUNTER_H

#ifdef __cplusplus
extern "C" {
#endif

#define COUNTER_EN       (1 << 0)
#define COUNTER_MODE    (1 << 1)

#define COUNTER_INPMODE   (COUNTER_MODE * 0)
#define COUNTER_RTIMODE   (COUNTER_MODE * 1)

typedef struct pacc {
   volatile unsigned int control;   // (r/w) reset conter
   volatile unsigned int counter;   // (r)
} counter_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/counter.h

20. Timer Watchdog Module (timer-wdt)

The timer watchdog module can be used for system monitoring purposes. Typically, the application has to clear the watchdog counter at regular intervals otherwise it generates an system reset or interrupt. The timer watchdog module requires two clock inputs: On the one hand CLK_1 which is used as timer input for the watchdog counter, on the other hand CLK_X which garantees the functionality of this module during system reset. (CLK_X has to be completly independend of the remaining SoC design.)
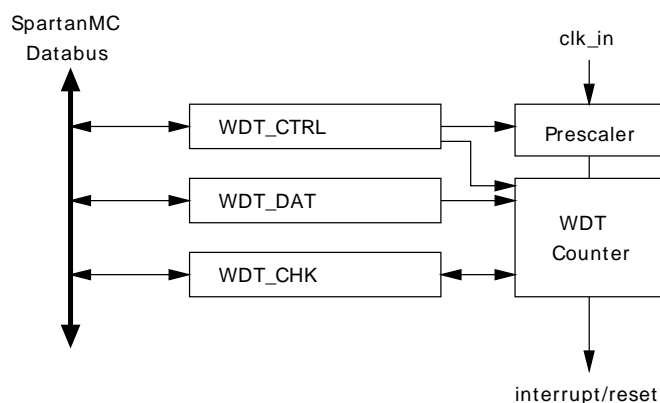
Figure 20-51: Watchdog timer block diagram

Note: The timer watchdog moudule can be used as stand alone peripheral or in connection with an basic timer module (used as counter clock input).

20.1. Usage

During the operation of the watchdog timer peripheral the value in WDT_DAT register is incremented continuously. If the value in WDT_DAT reaches a configured maximum value the peripheral performs a globel reset or an interrupt (with maximum priority). The reset of the watchdog counter is perfomend by writing a specific data word to WDT_CHK register. To determine the alert status of the watchdog module after a system reset bit 5 (ALARM bit) of WDT_CTRL has to be read .

20.2. Module Parameters

Table 20-68: Timer watchdog module parameters

Parameter	Default Value	Descripton
BASE_ADR		Start address of the memory mapped peripheral registers. The value is taken as offset to the start address of the peripheral memory space. **This parameter is set by jConfig automatically.**
WDT_RESET_PIN	0x12345	Code word to clear the watchdog timer.

20.3. Interrupts

If the watchdog counter reaches its maximum value an interrupt can be generated. The interrupt can be cleared by writing the WTD_CTRL register.

20.4. Peripheral Registers

20.4.1. Timer Watchdog Register Description

The timer watchdog peripheral provides three 18 bit registers which are mapped to the SpartanMC address space e.g. 0x1A000 + BASE_ADR + Offset.

Table 20-69: Timer watchdog registers

Offset	Name	Access	Description
0	WDT_CTRL	read/write	Specify the operation mode of the watchdog timer. (Each write access clears the ALARM bit)
1	WDT_DAT	read/write	Maximum value of watchdog timer.
2	WDT_CHK	read	If read it contains the current value of the watchdog timer.
2	WDT_CHK	write	**Clears the watchdog timer if written with the configured code word.**

20.4.2. WDT_CTRL Register

Table 20-70: WDT_CTRL register layout

Bit	Name	Access	Default	Description
0	WDT_EN	read/ write	0	If set to one the watchdog timer is enabled.
1	WDT_EN_PRE	read/ write	0	If set to one the prescaler is enabled.
2-4	WDT_PRE_VAL	read/ write	000	Specify the prescaler value : $000 = 2^1$ $001 = 2^2$ $010 = 2^3$ $011 = 2^4$ $100 = 2^5$ $101 = 2^6$ $110 = 2^7$ $111 = 2^8$
5	WDT_ALARM	read/ write	0	Determines a watchdog alert. Set to null on each write access to this register.
6-17	x	read	0	Not used.

Table 20-70: WDT_CTRL register layout

20.4.3. WDT_DAT Register

Table 20-71: WDT maximum value register layout

Bit	Name	Access	Default	Description
0-17	Max Counter	read/ write	x	Specify the maximum counter value.

20.4.4. WDT_CHK Register

Table 20-72: WDT counter register layout

Bit	Name	Access	Default	Description
0-17	Main Counter	read/ (write)	0	If read it contains the current watchdog counter value. If written with WDT_RESET_PIN it clears the watchdog counter value.

20.4.5. WDT C-Header for Register Description

```c
#ifndef __WATCHDOG_H
#define __WATCHDOG_H

#ifdef __cplusplus
extern "C" {
#endif

#define WATCHDOG_EN         (1<<0)
#define WATCHDOG_PRE_EN     (1<<1)
#define WATCHDOG_PRE_VAL    (1<<2)      // *0 fuer 2^1 bis *7
fuer 2^8

#define WATCHDOG_PRE_2      (WATCHDOG_PRE_VAL * 0)
#define WATCHDOG_PRE_4      (WATCHDOG_PRE_VAL * 1)
#define WATCHDOG_PRE_8      (WATCHDOG_PRE_VAL * 2)
#define WATCHDOG_PRE_16      (WATCHDOG_PRE_VAL * 3)
#define WATCHDOG_PRE_32      (WATCHDOG_PRE_VAL * 4)
#define WATCHDOG_PRE_64      (WATCHDOG_PRE_VAL * 5)
#define WATCHDOG_PRE_128   (WATCHDOG_PRE_VAL * 6)
#define WATCHDOG_PRE_256   (WATCHDOG_PRE_VAL * 7)

#define   WATCHDOG_ALARM      (1<<5)

typedef struct wdt {
   volatile unsigned int control;  // (r/w)
   volatile unsigned int limit;   // (r/w)
   volatile unsigned int val_rst;   // (r = val / w PIN = rst)
} watchdog_regs_t;
#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/watchdog.h

21. Universal Serial Bus v1.1 Device Controller (USB 1.1)

21.1. Overview

Das Interface wird mit einem auf 18 Bit Breite konfigurierten Blockram realisiert. Ein Port des Blockrams ist mit dem Systembus des SpartanMC verbunden und das zweite Port mit dem USB-Interface. Das Modul hat keine I/O-Register. Die Komunikation erfolgt nur durch Daten lese und schreib Zugriffe in diesen Blockram. Jeder der installierten Endpunkte kann einen Interrupt auslï¿½sen, wenn der Host Daten von einem IN-Endpunkt (Tx) gelesen hat oder wenn der Host Daten auf einen OUT-Endpunkt (Rx) abgelegt hat. Das Interface kann maximal 6 Endpunkte realisieren. An die drei erzeugten Signale ist nur noch folgende externe Beschaltung notwendig:

Figure 21-52: Der 1,5K Widerstand(external link) zieht D+ bei Disc=1 auf 3,3V wodurch das Interface im FULL-Speed Mode angemeldet wird.

21.2. Speicherorganisation

Die Basiadresse des USB Modul liegt oberhalb des Arbeitsspeichers der Konfiguration. Die Adressen (Offset) 0x000 bis 0x07f des DMA-Speichers sind fï¿½r die Konfiguration des USB-Interfaces reserviert. Im verbleibenden Bereich (0x080 bis 0x3ff) befinden sich 28 Datenspeicher mit je 32 Worten (64 Bytes). Sie werden je nach Konfiguration den Endpunkten zugeordnet. Bei deaktiviertem Double Buffering sind die Puffer aufeinander folgend den Endpunkten 0 bis 15 zugeordnet. Der Berich ab 0x280 bleibt unbenutzt. Ist Double Buffering aktiviert werden in aufsteigender Reihenfolge jedem der Endpunkte 0 bis 13 zwei Puffer zugeordnet. Enpunkt 14 und 15 kann nicht verwendet werden! Siehe Adresstabelle. Die aktuelle Implementierung der Hardware kann nur maximal 6 Endpunkte verwalten!

21.3. Konfigurations- und Statusregister

Offset	Register	Bemerkung
0x01	ep0c2	Globales Kommandoregister 2
0x02	ep1c	Kommandos fï¿½r Endpunkt 1
0x03	ep1s	Status von Endpunkt 1
0x04	ep2c	Kommandos fï¿½r Endpunkt 2
0x05	ep2s	Status von Endpunkt 2
...	...	...
0x1e	ep15c	Kommandos fï¿½r Endpunkt 15
0x1f	ep15s	Status von Endpunkt 15
0x20	glob	Globales Kommandoregister

Table 21-73: Die aktuelle Implementierung unterstï¿½tzt nur 6 Endpunkte!

21.4. Descriptoren (read only)

Offset	Bemerkung
0x21	Device Descriptor
0x2a	Configuration Descriptor
0x68	Language Descriptor
0x6a	String Descriptor describing manufacturer
0x73	String Descriptor describing product
0x7c	String Descriptor describing serial number

Table 21-74: Descriptoren

21.5. Puffer

Offset	Puffer	Bemerkung	EP ohne double buffering	EP mit double buffering
0x080	data00	Puffer 0 fï¿½r 64 Byte	0	0 (0)
0x0a0	data01	Puffer 1 fï¿½r 64 Byte	1	0 (1)
0x0c0	data03	Puffer 2 fï¿½r 64 Byte	2	1 (0)
0x0e0	data03	Puffer 3 fï¿½r 64 Byte	3	1 (1)
...	...	...	...	...
0x240	data14	Puffer 14 fï¿½r 64 Byte	14	7 (0)
0x260	data15	Puffer 15 fï¿½r 64 Byte	15	7 (1)
...	...	...	...	...
0x3c0	data26	Puffer 26 fï¿½r 64 Byte	26	13 (0)
0x3e0	data27	Puffer 27 fï¿½r 64 Byte	27	13 (1)

Table 21-75: Adressen der Puffer

Die aktuelle Implementierung unterstï¿½tzt nur 6 Endpunkte! Die Anordnung der Bytes in den Puffern kann mit dem Parameter NOGAP verï¿½ndert werden. Mit NOGAP=0 werden die Bytes in der 9 Bit Anordnung des SpartanMC abgelegt. Diese Anordnung ist fï¿½r die ï¿½bertragung von Zeichenketten sinnvoll. Sollen 16 Bit Werte vom SpartanMC in dem DMA-Puffer abgelegt werden, dann mï¿½ssen die Byte im 8 Bit Abstand in das 18 Bit Wort eingetragen Werden. Diese Anordnung der Bytes wird mit NOGAP=1 eingestellt.

Lesen eines 16 Bit Wortes aus einem Puffer mit NOGAP=0 oder NOGAP=1

```
// lesen 16 Bit
   unsigned int wert16;
#if SB_USB11_0_NOGAP == 0
   unsigned int i;
   unsigned int j;
   i = USB11_0_DMA->data02[0];
   // 2 SpMC Byte zu 16 Bit zusammen fassen
   j = i & 0x3fe00;
   i = i & 0x000ff;
   j = j >> 1;
   wert16 = j | i;
#else
   wert16 = USB11_0_DMA->data02[0];
#endif
```

Schreiben eines 16 Bit Wortes in einen Puffer mit NOGAP=0 oder NOGAP=1

```
// schreiben 16 Bit
   unsigned int wert16;
```

```
#if SB_USB11_0_NOGAP == 0
    unsigned int k;
    unsigned int l;
    k = wert16;
    // 16 Bit in der SpMC Bytanordnung in k bilden
    l = k & 0x3ff00;
    k = k & 0x000ff;
    l = l < < 1;
    k = l | k;
    USB11_0_DMA->data01[0] = k;
#else
    USB11_0_DMA->data01[0] = wert16;
#endif
```

21.6. Bitbelegung der Register

21.6.1. epXc Register

Bit	Bezeichn	Bedeutung	
6-0	Size	Anzahl zu sendender Bytes	Wert mit 0x3f maskieren (0 entspricht 64 Byte)
10-7	Reserviert		
11	bufsel	Auswahl des aktiven Puffers bei double buffering.	0=Unterer Puffer, 1=Oberer Puffer (im Speicherbereich)
12	in	Tx Endpunkt zum Senden von Daten zum Host	1=Endpunkttyp IN, sonst 0
13	out	Rx Endpunkt zum Empfangen von Daten von Host	1=Endpunkttyp OUT, sonst 0
14	control	Steuerinformationen des Interface	1=Endpunkttyp CONTROL, sonst 0
15	mode	Datentransfer	1=synchron, 0=asynchron
16	intr	enable Interrupt	1=Interrupt enable, 0=Interrupt disable
17	en	enable (HOST darf lesen bzw. schreiben)	EP IN: 1=Puffer enhï¿½llt Daten, EP OUT: 1=Puffer leer

Table 21-76: epXc Register

21.6.2. epXs Register (read only)

Bit	Bezeichnung	Bedeutung
6-0	Anzahl empfangender Bytes	Anzahl zu sendender Bytes

Bit	Bezeichnung	Bedeutung
10-7	not used	

<p align="center">**Table 21-77: epXs Register (read only)**</p>

21.6.3. Globales Steuerregister

Bit	Bezeichnung	Bedeutung
0	iep00	Impuls setzt Interrupt EP 0 zurï¿½ck
...	...	...
15	iep15	Impuls setzt Interrupt EP 15 zurï¿½ck
16	ep0ie	enable EP0 Interrupt
17	en	enable USB-Interface

<p align="center">**Table 21-78: Globales Steuerregister**</p>

Die usb_init.c zur Vorinitialisierung der USB DMA Puffer befindet sich in "./spartan-mc/lib_obj/src/peri".

Eine externe Dokumentation findet sich bei Open-Cores (http://opencores.org/websvn,listing?repname=usb&path=%2Fusb%2Ftrunk%2Frtl%2Fverilog%2F#path_usb_trunk_rtl_verilog_).

22. Display Controller

The display controller is a periphal SpartanMC device for driving several types of displays. It is possible to control either a segment based or a pixel based display. For resource optimization both parts are seperated into independent controller modules selectable from the jConfig device menu. For a segment based LCD, a special circuit is required for connecting the SpartanMC FPGA to the device (see later). The pixel based display requires a circuit including elements for controlling the backlight and contrast voltage. Below, both controller parts are described in detail, starting with the segment dispaly controller.

22.1. Controller for segment based displays

This module makes it possible to control a segment based display with a user defined number of digits and segments. The required memory for storing the digit's segment assignments is already included (its content depends on the defined settings). From those segment assigments the signals for the display's multiplex driving are generated. In case the display is a liquid crystal display (LCD), the differing signal sequence for driving LCDs is generated accordingly. This requires a corresponding circuit for connecting the display as shown in the figure below. There, the micro controller's output is connected in the center of a voltage divider with two equal resistors. The divider itself is connected to operating voltage and ground.
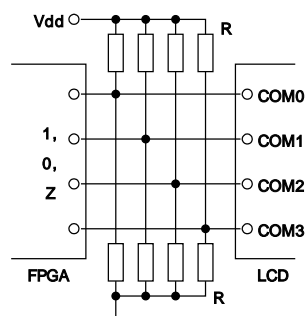
Figure 22-53: Circuit for connecting the LCD

If the display is not connected by using the given circuit, it will be damaged permanently! Because the dc voltage is not excluded.

22.1.1. Periphal registers

Table 22-79: Configuration registers of the segment display controller

Offset	Name	Description	Access	Initialization
0	REG_ENABLE	De-/Activates the display	read/ write	0x00000
1	REG_ADDR	Contains the adress of the digit to read from or write to.	read/ write	0x00000
2	REG_DATA	Contains the segment assignments according to the given digit address	read/ write	0x00000

Table 22-79: Configuration registers of the segment display controller

For accessing the segment memory the adress register must first be set to the correct digit number. Afterwards its current content can be read from or written to the data register.

22.1.2. Memory layout

The memory layout is slightly unconventional. This is caused by the required flexibility for configuring the number of segments/commons. Hence the data word is divided into the number of parts the display has commons. This allows the controller to compute the segment data sequentially for each common cycle instead of picking the required segment information out of the hole data word (which is quite complex to realize dynamically in hardware). The exact segment order depends on the used display. The first part of the data word (starting with bit 0) contains all assignments for the segments to be driven at common cycle 0, the next part for common cycle 1 and so on. For a 14 segment display with 4 commons the data word would look like "mPnd lcke gbjf haiS", where each letter represents one segment according to the typical segment order of such a display.

22.1.3. Module parameters

Table 22-80: Parameters of the segment display controller

Parameter	Description
SYSTEM_FREQUENCY	Clock the system is currently driven by (for example 25 MHz)

Parameter	Description
SEGMENT_FREQUENCY	Specifies the frequency for driving a single segment (see display's specification)
NUMBER_OF_COMMONS	Number of the display's common connections (number of anodes for LED displays)
NUMBER_OF_SEGMENTS	Number of segment connection for the whole display
NUMBER_OF_DIGITS	Number of the display's equal digits
BIT_PER_DIGIT	Width of a single memory word inside the segment memory (usually equal the the number of segments per digit)
IS_LCD	Set to 1 for a LCD, 0 for a LED display

Table 22-80: Parameters of the segment display controller

22.2. Controller for pixel based displays

This part of the display controller allows the driving of almost any pixel based displays. Therefore it provides the required memories. Depending on the module's configuration the operation of a graphic and a text mode is possible whereas both modes run independly. Inside the text mode a blinking cursor is displayed whose appearance can be defined by the user. The required codepage for converting the character codes to according pixel data can also be changed by the user. By default the codepage is initialized during the synthesis of the design with the "codepage 437", known from the original IBM PC. This initialization is defined in the given user constraint file (UCF). Inside the graphic mode there are several hardware accelerated functions for accessing the video memory (e.g. SetPixel or Line).

22.2.1. Periphal registers

Table 22-81: Configuration register of the matrix display controller

Offset	Name	Description	Access	Initialization
0	REG_DISPLAY_STATUS	Status register of the whole controller (see below)	read/write	0x00100
1	REG_TEXT_COLOR	Foreground color of the displayed text. This color must come from the display's color space.	read/write	0x0000F (white)
2	REG_TEXT_BGCOLOR	Background color of the displayed text	read/write	0x00000

Offset	Name	Description	Access	Initialization
3	REG_TEXT_CHARPOS	Contains the coordinates of the currently drawed character. This may be important in relation to the CharLine interrupt described later.	read	0x00000
4	REG_TEXT_CURSORPOS	Position of the blinking cursor in text mode	read/ write	0x00000
5	REG_GRAPH_COORDSELECT	Register for addressing a single coordinate for the graphic functions	read/ write	0x00000
6	REG_GRAPH_COORDVALUE	Value of the selected coordinate	read/ write	0x00000
7	REG_GRAPH_COLOR	Color for a graphic function. This is a color coming from the color space of the memory (color LUT)	read/ write	0x00000

Table 22-81: Configuration register of the matrix display controller

22.2.2. Assembly of the register REG_DISPLAYSTATUS

Table 22-82: Register REG_DISPLAYSTATUS

Bit	Name	Description	Access	Initialization
0	STATUSBIT_DISP_BACKLIGHT	Turns the backlight on or off.	read/ write	0
1	STATUSBIT_DISP_ON	De-/activates the display.	read/ write	0
2	STATUSBIT_TEXTMODE	De-/activates the text mode, if implemented.	read/ write	0
3-5	STATUSBIT_FUNCTION_SELECT	Selects a hardware accelerated graphic function.	read/ write	000
6	STATUSBIT_FUNCTION_FLAG1	Optional flag for the graphic functions	read/ write	0
7	STATUSBIT_FUNCTION_DRAW	Starts the selected graphic function with the given parameters. After	read/ write	0

Bit	Name	Description	Access	Initialization
		setting the bit, it will be reset in the next clock cycle.		
8	STATUSBIT_FUNCTION_READY	Indicated if the currently selected graphic function is ready.	read	1
9	STATUSBIT_OVERLAY	De/activates the overlay	read/ write	0

Table 22-82: Register REG_DISPLAYSTATUS

22.2.3. Assembly of REG_TEXT_CHARPOS and REG_TEXT_CURSORPOS

Table 22-83: Registers REG_TEXT_CHARPOS and REG_TEXT_CURSORPOR

Bit	Name	Description
0 to 8	Y	Y coordinate in the text mode
9 to 17	X	X coordinate in the text mode

Table 22-83: Registers REG_TEXT_CHARPOS and REG_TEXT_CURSORPOR

22.2.4. Interrupts

Table 22-84: Interrupts of the matrix display controller

Interrupt	Description
charLine_ir	Indicates that the drawing of one charcter's pixel line in text mode has completed. This makes it possible to change the color settings for the following character lines for example.
graphFunctionReady_ir	This interrupt is triggered when a graphic function gets ready.

Table 22-84: Interrupts of the matrix display controller

22.2.5. Coding of the graphic functions

The following coding is defined in the file "display_graph_common.v" and should be changed there if required.

Table 22-85: Implemented graphic functions

Number	Function	Macro name
0	invalid	-
1	SetPixel	FUNCTION_SETPIXEL
2	Line	FUNCTION_LINE
3	CopyRect	FUNCTION_COPYRECT
4	GetPixel	FUNCTION_GETPIXEL
5	FillRect	FUNCTION_FILLRECT

Table 22-85: Implemented graphic functions

22.2.6. Memory layouts

Codepage The codepage is a continous memory containing the pixel data of every displayable character. One memory word contains a single pixel line of a character. Thus one character requires a certain number of memory words depending on the configuration (by default 16). The memory offset O of one character is calculated by O = C * H, where C is the character code an H the configured number of lines per character.

Text cursor The layout of the cursor memory is equal to the one of the codepage but contains space for only one character.

Graphic memory Depending on the configuration one word of the graphic memory contains data for several pixels. Therein the MSB represents the most left and the LSB the most right pixel (almost like little endian). Usually the user has no need to access the graphic memory directly since there are functions like SetPixel and GetPixel.

Color LUT The color look up table (color LUT) converts the reduced color space inside the graphic memory to the display's color space. Thus the offset inside the color LUT represents a color of the graphic memory. From this offset the color LUT offers the corresponding color for the display.

22.2.7. Module parameters

Table 22-86: Parameters of the matrix display controller

Name	Description
BASE_ADDR	Base address on which the controller communicates with the SpartanMC
GRAPHMEM_BASE_ADDR	Base address on which the graphic memory is connected. The memory should be placed behind the space for I/O devices in any case. This is caused through its size, where otherwise some I/O device may be activated accidentally.

Name	Description
TEXTMEM_BASE_ADDR	Base address of the text memory
CODEPAGE_BASE_ADDR	Base address of the codepage
COLOR_LUT_BASE_ADDR	Base address of the color LUT
CURSORMEM_BASE_ADDR	Base address of the text cursor's memory
SCREEN_WIDTH	Display's width in pixels
SCREEN_HEIGHT	Display's height in pixels
DATA_WORD_WIDTH	Width of one data word transmitted to the display
BIT_PER_PIXEL	Color depth of one pixel on the display
BIT_PER_MEMORY_PIXEL	Color depth of one pixel in the graphic memory
TIMING_INIT_CLOCKS_TO_VCON_ON	Number of clocks to wait after a reset until the display's contrast voltage is activated.
TIMING_INIT_CLOCKS_TO_DISPOFF	Number of clocks to wait after a reset until the display itself is activated.
TIMING_CL1_CLOCKS_AFTER_RESET	Number of clocks to wait after a reset until a row cycle begins.
TIMING_CL1_CLOCKS_HIGH	Number of clocks the row clock is high.
TIMING_CL1_CLOCKS_LOW	Number of clocks the row clock is low.
TIMING_CL1_BEGIN_HIGH_LOW	Indicates whether a row on the display starts with the falling (1) or the rising edge (0) of the row clock.
TIMING_ROW_BREAK_DELAY	Number of clocks to wait after transmitting one row before the transmission of the next row starts.
TIMING_SCREEN_FINISH_DELAY	Number of clocks to wait after transmitting one complete screen before the transmission of the next screen continues.
TIMING_CL2_CLOCKS_AFTER_RESET	Number of clocks to wait after a reset until a data clock cycle begins.
TIMING_CL2_CLOCKS_HIGH	Number of clocks the data clock is high.
TIMING_CL2_CLOCKS_LOW	Number of clocks the data clock is low.
TIMING_FRAMESTART_CLOCKS_AFTER_RESET	Number of clocks to wait after a reset until the framestart cycle begins.
TIMING_FRAMESTART_CLOCKS_HIGH	Number of clocks the framestart signal is high.
TIMING_FRAMESTART_CLOCKS_LOW	Number of clocks the framestart signal is low.
TIMING_INVERT_CLOCKS_AFTER_RESET	Number of clocks to wait after a reset until the invert signal cycle begins.
TIMING_INVERT_CLOCKS_TOGGLE	Number of clocks after the invertsignal is inverted.
CODEPAGE_CHAR_WIDTH	Width of one character in pixles
CODEPAGE_CHAR_HEIGHT	Height of one character in pixels

Name	Description
CODEPAGE_SIZE	Number of characters in the codepage
FPGA_BRAM_SIZE	Number of words the used Block RAM can save
USE_TEXT_MODE	Defines whether the text mode is included in synthesis.
USE_GRAPH_MODE	Defines whether the graphic mode is included in synthesis.
USE_GRAPHFUNCTION_SETPIXEL	Defines whether the graphic function "SetPixel" is included in synthesis.
USE_GRAPHFUNCTION_LINE	Defines whether the graphic function "Line" is included in synthesis.
USE_GRAPHFUNCTION_COPYRECT	Defines whether the graphic function "CopyRect" is included in synthesis.
USE_GRAPHFUNCTION_GETPIXEL	Defines whether the graphic function "GetPixel" is included in synthesis.
USE_GRAPHFUNCTION_FILLRECT	Defines whether the graphic function "FillRect" is included in synthesis.

Table 22-86: Parameters of the matrix display controller

23. Core connector for multicore systems

The core connector implements a simple FIFO through which two SpartanMC cores are able to communicate. Therefor the modules master core connector as data transmitter and slave core connector as data receiver are available for unidirectional communication. The module duplex core connector provides an interface for bidirectional communication containing one master and one slave core connector.
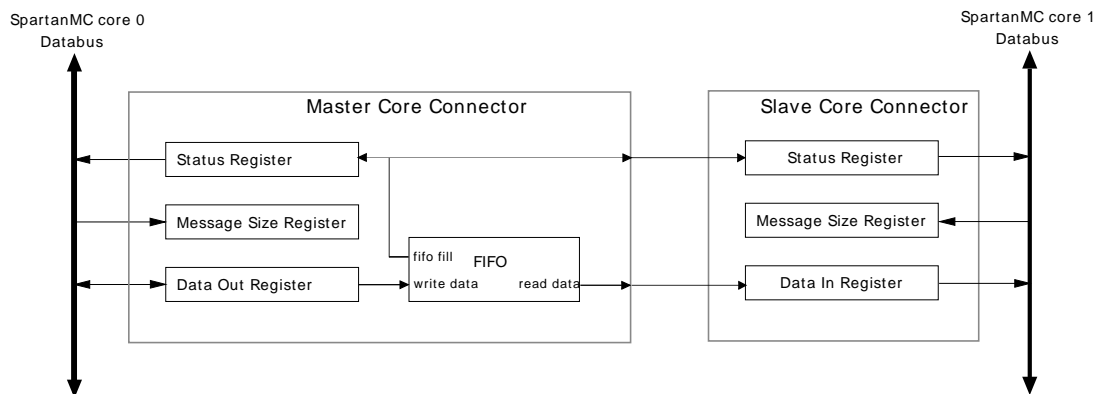
Figure 23-54: Unidirectional core connector

23.1. Module Parameters

Parameter	Default Value	Descripton
FIFO_WIDTH	18	FIFO width of the connector. It is highly recommendet to use the default value for optimal utilization of the FPGA structures.
FIFO_DEPTH	16	Amount of buffer registers used.

Table 23-87: Module parameters

23.2. Peripheral Registers

The slave and master core connectors have three registers each for message transfer:

23.2.1. STATUS Register Description

The status register tells if the FIFO is empty or full and if there is enough space left for another message. The possible return arguments are as follows:

Value	Description
0	FIFO has at least MSG_SIZE entries free.
1	FIFO is empty or fewer entries than specified by MSG_SIZE are used.
2	FIFO is full or has fewer free entries than specified in the MSG_SIZE register.

Table 23-88: STATUS states

23.2.2. MSG_SIZE Register Description

In the message size register should be written the size of the message to be written/read. It is needed by the status register to correctly signal whether the FIFO is full or empty. It does not influence the written or read data. Set to 1 by default.

23.2.3. DATA_OUT Register Description

This register is used by the master core connector to write data in. The data is handed to the slave core connectors data in register.

23.2.4. DATA_IN Register Description

This register is used by the slave core connector to read data from the master core connector.

23.3. Usage examples: MPSoC Lib

To communicate at a higher abstraction level the mpsoc library (see spartanmc/include/mpsoc.h) can be used. It offeres various functions for blocking and non-blocking send and receive for all inter core communication peripherals. To use the library add mpsoc to LIB_OBJ_FILES in config-build.mk in the firmware folder.

The _nb specifies non blocking funtions. Meaning that the function sends/receives the data if possible, if not it returns an error code (CORE_CONN_FULL / CORE_CONN_EMPTY). The functions also have a _value addition for sending one value or _data for sending more data, like arrays. The _data functions receive/send data in blocks of 16. In ideal case the buffer depth should be multiples of 16. The message to be send can also be larger than the buffer size. If the core connector is read from both sides at the same time via the _data functions the buffer depth shall ideally be 32.

23.3.1. Minimal send example

```
#include <system/peripherals.h>
#include <peripherals/duplex_core_connector.h>
#include <mpsoc.h>

master_core_connector_regs_t *master= DUPLEX_CORE_CONNECTOR_0;

int main(){
    int a[5]={1,2,3,4,5};
    core_connector_master_send_value(master,5);
    core_connector_master_send_data(master,&a, 5);
    while(1);
}
```

23.3.2. Minimal receive example

```
#include <system/peripherals.h>
#include <peripherals/duplex_core_connector.h>
#include <mpsoc.h>

slave_core_connector_regs_t *slave= DUPLEX_CORE_CONNECTOR_0 +
DUPLEX_CONNECTOR_SLAVE_OFFSET;

int main(){
    int size=core_connector_slave_receive_value(slave);
    int a[size];
    core_connector_slave_receive_data(slave,&a, size);
    while(1);
}
```

24. Concentrator system for multicore systems

The concentrator system implements a N to 1 connection between multiple SpartanMC cores. Several slaves can transmit data to the master core unidirectionaly. An internal round robin arbiter implements the selection of the slaves.

24.1. Module Parameters

24.1.1. Master

Parameter	Default Value	Descripton
BUFFER_SIZE	64	Size of the integrated FIFO buffer.
NUMBER_OF_SLAVES	3	Number of slaves connected to the master.
SOFTWARE_ARBITRATION	0	Defines if direct software arbitration is used instead of the round robin arbiter.

Table 24-89: Master module parameters

24.1.2. Slave

Parameter	Default Value	Descripton
SLAVE_ID	0	Unique ID of the slave.

Table 24-90: Slave module parameters

24.2. Peripheral Registers

24.2.1. Master

The master offers the following registers.

Name	Description
STATUS	Returns if the amount of data depicted by MSG_SIZE is available in the buffer.

Name	Description
MSG_SIZE	Can only be written to check the buffer (see STATUS register).
DATA_OUT	Register to read out the data values and header.
PEEK_DATA	Register to read out the header packet without affecting the fifo's content.
SOFT_ARBIT	Control register to define the slave id that is granted access in case of direct software arbitration.

Table 24-91: Registers

24.2.2. Register usage

First, set MSG_SIZE to "1" to check if there is any data in the buffer. This can be done by reading out the STATUS register. If it returns BUFFER_EMPTY there is no data to be read. Else, the header can be read from the DATA_OUT register. The 9 LSB of the header contain the source slave ID, the MSB of the header contain the message size. According to the message size, data can be read from the DATA_OUT register.

24.2.3. Slave

The slave offers the following.

Name	Description
STATUS	Returns if enough space is left in buffer for amount of data depicted by MSG_SIZE.
MSG_SIZE	Can only be written to check the buffer (see STATUS register).
DATA_IN	Register to write data values and header.
DATA_AVAILABLE	Register to mark the availability of data.

Table 24-92: Registers

24.2.4. Register usage

First, write the message size plus one (for the header) to the MSG_SIZE register and set DATA_AVAILABLE to "1". Then, read the STATUS register to check if there is enough space in the buffer. If the STATUS register contains NO_SEND_PERMISSION, this slave is not permitted to send data. If it contains BUFFER_FULL, the slave is permitted to send data, but there is not enough space in the buffer for the whole message. Use the copy_mem_to_reg_with_small_buffer function then. If it contains BUFFER_AVAILABLE, the whole message can be written to DATA_IN at once. Finally, set DATA_AVAILABLE to "0".

24.3. Usage examples

24.3.1. Register level access

Because of the complex register usage, it is highly recommended to use the C library which can be included by adding "#include <concentrator.h>" at the top of the program. The functions of the library use the high efficiency copy functions of "mpsoc.c".

24.3.2. Slave - sending a packet with the blocking function

Sending a package with 5 values

```
unsigned int data_array[5] = {1, 2, 3, 4, 5};
unsigned int number_of_values = 5;

concentrator_slave_send_data(SLAVE_CONCENTRATOR_0,
&data_array, number_of_values);
```

24.3.3. Slave - sending a packet with the non-blocking function

Sending a package with 5 values

```
unsigned int data_array[5] = { 1, 2, 3, 4, 5 };
unsigned int number_of_values = 5;

concentrator_slave_data_available_request_for_nb(SLAVE_CONCENTRATOR_0,
number_of_values);
while (concentrator_slave_send_data_nb(SLAVE_CONCENTRATOR_0,
&data_array, number_of_values) != CONCENTRATOR_SEND_OK);
```

24.3.4. Master - receiving a packet with the blocking function

```
unsigned int data_array[50];
unsigned int number_of_values = 0, slave_id = 0;

concentrator_slave_send_data(MASTER_CONCENTRATOR_0,&data_array,
&number_of_values, &slave_id);
```

24.3.5. Master - receiving a packet with the non-blocking function

```
unsigned int data_array[50];
unsigned int number_of_values = 0, slave_id = 0;
```

```
while
(concentrator_slave_send_data_nb(MASTER_CONCENTRATOR_0,&data_array,
&number_of_values, &slave_id) != CONCENTRATOR_RECEIVE_OK);
```

25. Dispatcher system for multicore systems

The dispatcher system implements a 1 to N connection between multiple SpartanMC cores. A master can transmit data to several slave cores unidirectionally. Packages can be sent to individual receivers or can be distributed via an internal round robin arbiter or a load balancing arbiter, which chooses the slave with the fewest number of waiting data entries in the buffer.

25.1. Module Parameters

25.1.1. Master

Parameter	Default Value	Descripton
BUFFER_SIZE	64	Size of the integrated FIFO buffer.
NUMBER_OF_SLAVES	3	Number of slaves connected to the master.

Table 25-93: Master module parameters

25.1.2. Slave

Parameter	Default Value	Descripton
SLAVE_ID	0	Unique ID of the slave.

Table 25-94: Slave module parameters

25.2. Peripheral Registers

25.2.1. Master

The master offers the following registers.

Name	Description
STATUS	Returns if enough space is left in buffer for amount of data depicted by MSG_SIZE.

Name	Description
MSG_SIZE	Can only be written to check the buffer (see STATUS register).
DATA_IN	Register to write data values.

Table 25-95: Master registers

25.2.2. Register usage

Write the amount of data to be written plus one (for the packet header) to the MSG_SIZE register. Accordingly, the STATUS register returns BUFFER_AVAILABLE or BUFFER_FULL. If the return value is BUFFER_AVAILABLE, the whole message can be written to the DATA_IN register at once. If the return value is BUFFER_FULL, the message can be written in pieces (e.g. using the copy_mem_to_reg_with_small_buffer function) or the send action can be aborted.

25.2.3. Slave

The slave offers the following registers.

Name	Description
STATUS	Returns if amount of data depicted by MSG_SIZE is available for this slave.
MSG_SIZE	Can only be written to check the buffer (see STATUS register).
DATA_OUT	Register to read out the data values.
MSG_SIZE_IN	Contains the amount of data of the current message.
RECEIVED	Can be written to confirm the complete reception of the message.

Table 25-96: Slave registers

25.2.4. Register usage

Set MSG_SIZE to "1" to check if any data is available for the slave by reading out the STATUS register. If DATA_AVAILABLE is returned, data is available. Then, the MSG_SIZE_IN register can be read to obtain the size of the message to be read. To check whether the full message is present in the buffer, this read out message size can be written to the MSG_SIZE register. If the STATUS register returns DATA_AVAILABLE, the whole message can be read from the DATA_OUT register at once. If the STATUS register returns BUFFER_EMPTY, the message has to be read out in pieces (e.g. using the copy_reg_to_mem_with_small_buffer function). Finally, the RECEIVED register has to be written to confirm the complete reception of the message.

25.3. Usage examples

25.3.1. Register level access

Because of the complex register usage, it is highly recommended to use the C library which can be included by adding "#include <dispatcher.h>" at the top of the program. The functions of the library use the high efficiency copy functions of "mpsoc.c".

25.3.2. Master - sending a packet with the blocking function

Sending a package with 5 values.

```
unsigned int data_array[5] = {1, 2, 3, 4, 5};
unsigned int number_of_values = 5;

//Receiver id = 2
dispatcher_master_send_data(MASTER_DISPATCHER_0, &data_array,
number_of_values, 2);

//Receiver id choosen by round-robin arbiter
dispatcher_master_send_data(MASTER_DISPATCHER_0, &data_array,
number_of_values, DISPATCH_ROUND_ROBIN);

//Receiver id choosen by load-balancing arbiter
dispatcher_master_send_data(MASTER_DISPATCHER_0, &data_array,
number_of_values, DISPATCH_LOAD_BALANCING);
```

25.3.3. Master - sending a packet with the non-blocking function

Sending a package with 5 values.

```
unsigned int data_array[5] = {1, 2, 3, 4, 5};
unsigned int number_of_values = 5;

//Receiver id = 2
while ( dispatcher_master_send_data_nb(MASTER_DISPATCHER_0,
&data_array, number_of_values, 2) != DISPATCHER_SEND_OK);

//Receiver id choosen by round-robin arbiter
while ( dispatcher_master_send_data_nb(MASTER_DISPATCHER_0,
&data_array, number_of_values, DISPATCH_ROUND_ROBIN) !=
DISPATCHER_SEND_OK);

//Receiver id choosen by load-balancing arbiter
```

```
while ( dispatcher_master_send_data_nb(MASTER_DISPATCHER_0,
&data_array, number_of_values, DISPATCH_LOAD_BALANCING) !=
DISPATCHER_SEND_OK);
```

25.3.4. Slave - receiving a packet with the blocking function

```
unsigned int data_array[50];
unsigned int number_of_values = 0;

dispatcher_slave_read_data(SLAVE_DISPATCHER_0, &data_array,
&number_of_values);
```

25.3.5. Slave - receiving a packet with the non-blocking function

```
unsigned int data_array[50];
unsigned int number_of_values = 0;

while ( dispatcher_slave_read_data_nb(SLAVE_DISPATCHER_0,
&data_array, &number_of_values) != DISPATCHER_RECEIVE_OK);
```

26. Real Time Operating System

The SpartanMc project contains a small Real Time Operating System that can be used to easily run multiple tasks in parallel. The tasks are scheduled based on their priority and can be synchronized by using semaphores.

26.1. Concepts

To avoid having to save all registers a task uses, every task is assigned a part of the register file. The size of this part has to be set at task creation by specifying the number of call levels in the task. Additionally, every task is assigned a part of memory for its stack.

Note: Neither stack nor registers are range checked. It is up to the application programmer to ensure that a task does not overwrite data outside of its assigned resources.

Scheduling is based on priorities. When a task with higher priority than the currently running task gets ready, it is immediately scheduled.

Note: Do not switch tasks from within an ISR when using the complex interrupt controller. In that case the end of the ISR is not correctly signaled to the interrupt controller, which makes it ignore any future interrupts of the same or lower priority.

26.2. Preparing the Firmware

To link the RTOS into the firmware, edit the file *config-build.mk* in the firmware directory. In the list of libraries, remove *startup* and add *rtos* . If interrupt support is needed, also add *rtos_interrupt* .

The main source file needs to define these *int* variables to tell the RTOS how to initialize the main task:

Table 26-97: Needed variables for initialization of RTOS

Variable	Description
main_task_priority	Priority of the main task
main_task_max_call_level	Maximum call level in the main task
main_task_stack_size	Maximum stack size in the main task
isr_max_call_level	Maximum call level in interrupt service routines
isr_max_stack_usage	Maximum stack size in interrupt service routines

26.3. Task management

26.3.1. create_task

```
task_t create_task( void ( *entry ) (void * param), void
*param, uint18_t priority, uint18_t max_call_level, size_t
stack_size )
```
Create a new task. Returns a representation of the newly created task or *NULL* if the operation failed due to insufficient memory or insufficient free space in the register file.

Table 26-98: Parameters of create_task

Parameter	Description
entry	Entry Point of the task to create
param	Parameter to call *entry* with
priority	Priority of the task to create
max_call_level	Maximum call level for the task to create. Note that interrupts occurring during the execution of the task also need to be counted. The entry function itself is not to be counted in this value.
stack_size	Stack size of the task to create

Table 26-99: Info about create_task

Property	Value
Callable by ISR	No
Internal call depth	5
with active interrupts	4

26.3.2. delete_task

```
void delete_task( task_t task )
```
Deletes a task. If a task wants to delete itself, its memory is not freed immediately but later when the idle task is scheduled.

Table 26-100: Parameters of delete_task

Parameter	Description
task	The task to delete, as returned by create_task

Table 26-101: Info about delete_task

Property	Value
Callable by ISR	No
Internal call depth	5
with active interrupts	4

26.3.3. suspend_task

void suspend_task(task_t task)
Suspend a task. It can be resumed by calling resume_task.

Table 26-102: Parameters of suspend_task

Parameter	Description
task	The task to suspend, as returned by create_task

Table 26-103: Info about suspend_task

Property	Value
Callable by ISR	No
Internal call depth	3
with active interrupts	2

26.3.4. resume_task

void resume_task(task_t task)
Resume a previously suspended task.

Table 26-104: Parameters of resume_task

Parameter	Description
task	The task to resume, as returned by create_task

Table 26-105: Info about resume_task

Property	Value
Callable by ISR	No
Internal call depth	3

Property	Value
with active interrupts	2

26.3.5. get_current_task

`task_t get_current_task( )`
Return a representation of the current task.

Table 26-106: Info about get_current_task

Property	Value
Callable by ISR	Yes
Internal call depth	1
with active interrupts	1

26.3.6. forbid_preemption

`void` `forbid_preemption( )`
Forbid the preemption of the current task to mark critical sections. Event tasks with higher priority than the current task will not get scheduled until permit_preemption is called. Multiple calls to forbid_preemption are allowed to be able to nest critical sections. To allow preemption again, permit_preemption has to be called the same number of times.

Table 26-107: Info about forbid_preemption

Property	Value
Callable by ISR	No
Internal call depth	1
with active interrupts	1

26.3.7. permit_preemption

`void` `permit_preemption( )`
Permit the preemption of the current task. To allow preemption again, permit_preemption has to be called the same number of times as forbid_preemption.

Table 26-108: Info about permit_preemption

Property	Value
Callable by ISR	No
Internal call depth	1
with active interrupts	1

26.3.8. task_yield

void task_yield( )
Change to another task of the same priority if one is available.

Table 26-109: Info about task_yield

Property	Value
Callable by ISR	Yes
Internal call depth	3
with active interrupts	1

26.4. Semaphores

26.4.1. initialize_semaphore

task_t create_task(semaphore_t *sem, uint18_t value)
Initializes a semaphore. This function has to be called before using a semaphore for the first time. It cannot be used to change a semaphore's value while it is in use.

Table 26-110: Parameters of initialize_semaphore

Parameter	Description
sem	Pointer to the semaphore to initialize
value	The semaphore's initial value

Table 26-111: Info about initialize_semaphore

Property	Value
Callable by ISR	Yes (?)
Internal call depth	2
with active interrupts	2

26.4.2. semaphore_down

void semaphore_down(semaphore_t *sem)
Reduces the semaphore's value by one. If it already is zero, block the task until another task calls semaphore_up.

Table 26-112: Parameters of semaphore_down

Parameter	Description
sem	The semaphore

Table 26-113: Info about semaphore_down

Property	Value
Callable by ISR	No
Internal call depth	3
with active interrupts	2

26.4.3. semaphore_up

void semaphore_up(semaphore_t *sem)
Increases the semaphore's value by one. If there are threads blocked by this semaphore, wake one of them.

Table 26-114: Parameters of semaphore_up

Parameter	Description
sem	The semaphore

Table 26-115: Info about semaphore_up

Property	Value
Callable by ISR	No
Internal call depth	3
with active interrupts	1

26.5. Dynamic memory allocation

26.5.1. malloc

void *malloc(size_t size)
Allocate a block of memory.

Table 26-116: Parameters of malloc

Parameter	Description
size	the number of 9-bit-words to allocate.

Table 26-117: Info about malloc

Property	Value
Callable by ISR	No
Internal call depth	4
with active interrupts	3

26.5.2. free

void free(**void** *ptr)
Free a previously allocated block of memory.

Table 26-118: Parameters of free

Parameter	Description
ptr	Pointer to the memory block to free.

Table 26-119: Info about free

Property	Value
Callable by ISR	No
Internal call depth	4
with active interrupts	3

26.6. Example Code

This example code creates two threads that both output to an uart_light.

The hardware for this example is the same as in the quickstart guide: processor core, sysclk, and uart_light.

```c
#include <system/peripherals.h>
#include <uart.h>
#include <stdio.h>
#include <rtos.h>

int main_task_priority = 1, main_task_max_call_level =
10, main_task_stack_size = 200, isr_max_call_level = 10,
isr_max_stack_usage = 200;

static void hello_task (void * param) {
    while (1) {
        printf("hello from task %d\n",param);
        task_yield();
    }
}
FILE * stdout = &UART_LIGHT_0_FILE;

void main( void ) {

    create_task(hello_task, 1, 1, 5, 100);
    create_task(hello_task, 2, 1, 5, 100);

    suspend_task(get_current_task());
}
```

The line for linked libraries in *config-build.mk* looks like this:

```
LIB_OBJ_FILES:=rtos peri
```

27. Simple technology agnostic clock generator

The Simple technology agnostic clock generator provides a configurable output clock that is generated from an input clock. The input clock can be multiplied and divided to meet the user desired output frequency. It can be used for xilinx and altera projects.

The output frequency is calculated by the formula:

output frequency = (input frequency * MULTIPLY) / DIVIDE

27.1. Module Parameters

Table 27-120: Simple technology agnostic clock generator module parameters

Parameter	Descripton
CLKIN_PERIOD	The duration of one period of the input frequency in nanoseconds.
MULTIPLY	The multiplier for the input frequency. Supported values range from 2 to 32.
DIVIDE	The divider for the input frequency. Supported values range from 1 to 32.

28. Altera Cyclone 4 PLL

The Altera Cyclone 4 PLL clock generator provides up to five configurable output clocks which are generated from an input clock. The input clock can be multiplied and divided to meet the user desired output frequency. The output clocks can also be shifted.

This module aims to provide all important options of the altpll megafunction to the user. Not all combinations of MULTIPLY_BY, DIVIDE_BY and PHASE_SHIFT might be possible. If the synthesis fails with PLL related errors use the Quartus IDE to check if the combination of selected parameters is supported. To do this, create a Quartus project with device EP4CE22F17C6 and select "PLL" from the ip catalog. Configure the pll parameters in MegaWizard according to the settings in jconfig (all output clocks in MegaWizard tab 3 enabled, all parameters in MegaWizard tabs 1 and 2 default) and check if Megawizard says "Able to implement the requested PLL". If not, change the parameters in MegaWizard until the PLL is implementable and apply those settings to jconfig.

The output frequency for each output clock is calculated by the formula:

output frequency = (input frequency * MULTIPLY) / DIVIDE

28.1. Module Parameters

Table 28-121: Cyclone 4 PLL module parameters

Parameter	Descripton
INCLK0_INPUT_FREQUENCY	The frequency on the INCLK0 input in picoseconds.
CLKx_DIVIDE_BY	The divider for the input frequency of output clock x.
CLKx_MULTIPLY_BY	The multiplier for the input frequency of output clock x.
CLKx_DUTY_CYCLE	The duty cycle for output clock x.
CLKx_PHASE_SHIFT	The phase shift for output clock x.

29. Lattice VersaECP5 DevKit PLL

The Lattice VersaECP5 DevKit PLL provides a configurable output clock that is generated from an input clock. The input clock can be multiplied and divided to meet the user desired output frequency.

Because the parameters of the EHXPLLL primitive used on ECP5 devices are not explained in the lattice documentation and their calculation is dark magic hidden in the Lattice Diamond Clarity Designer GUI this module uses several hardcoded PLL implementations as an ugly workaround. The module expects a 100 MHz clock as input which is available on the VersaECP5 board at FPGA pin P3. The implementation to use for the project is chosen by the MULTIPLY parameter.

For most projects this should be sufficient and the use of Clarity Designer can be avoided. If more advanced PLL options are needed, please use the Module ECP5_PLL and get the necessary parameters from Clarity Designer.

The output frequency is calculated by the formula:

output frequency = (input frequency * MULTIPLY) / DIVIDE

29.1. Module Parameters

Table 29-122: Lattice VersaECP5 DevKit PLL module parameters

Parameter	Descripton
MULTIPLY	The multiplier for the input frequency. Supported values range from 4 to 100.
DIVIDE	The divider for the input frequency. Hardcoded to 100.

30. Lattice ECP5 PLL

The Lattice ECP5 PLL clock generator provides up to three configurable output clocks which are generated from an input clock. The input clock can be multiplied and divided to meet the user desired output frequency. The output clocks can also be shifted.

This module aims to provide all important options of the EHXPLLL primitive to the user. Because the parameters of the EHXPLLL primitive used on ECP5 devices are not explained in the lattice documentation and their calculation is dark magic hidden in the Lattice Diamond Clarity Designer GUI it is required to get the correct parameters from Clarity Designer.

To do this, create a Lattice Diamond project with device LFE5UM-45F-8BG381C (FPGA used on the VersaECP5 DevKit) and open Clarity Designer. Select "pll" from the ip catalog. Select "Int_OP" for "Feedback Mode" and 10% tolerance for CLKOP. Enter the desired phase shift and output frequency for CLKOS, CLKOS2 and CLKOS3 and click "calculate". Enter the parameters in jconfig.

For no phase shift, use CLKOS/2/3_DIV - 1 for CLKOS/2/3_CPHASE and 0 for CLKOS/2/3_FPHASE. If you need phase shift, generate the configured pll and open the generated verilog file with the EHXPLLL instance. Enter the CPHASE and FPHASE parameters in jconfig as set in the verilog file.

30.1. Module Parameters

Table 30-123: Lattice ECP5 PLL module parameters

Parameter	Descripton
CLKI_DIV	The Refclk Divider in the Diamond Clarity Designer UI.
CLKFB_DIV	The Feedback Divider in the Diamond Clarity Designer UI.
CLKOP_DIV	The CLKOP Output Divider in the Diamond Clarity Designer UI.
CLKOS_DIV	CLKOS Output Divider in the Diamond Clarity Designer UI.
CLKOS_CPHASE	CLKOS_CPHASE in the verilog file generated by Diamond Clarity Designer. Choose CLKOS_DIV - 1 for no phase shift.
CLKOS_FPHASE	CLKOS_FPHASE in the verilog file generated by Diamond Clarity Designer. Choose 0 for no phase shift.
CLKOS2_DIV	CLKOS2 Output Divider in the Diamond Clarity Designer UI.
CLKOS2_CPHASE	CLKOS2_CPHASE in the verilog file generated by Diamond Clarity Designer. Choose CLKOS2_DIV - 1 for no phase shift.
CLKOS2_FPHASE	CLKOS2_FPHASE in the verilog file generated by Diamond Clarity Designer. Choose 0 for no phase shift.
CLKOS3_DIV	CLKOS3 Output Divider in the Diamond Clarity Designer UI.
CLKOS3_CPHASE	CLKOS3_CPHASE in the verilog file generated by Diamond Clarity Designer. Choose CLKOS3_DIV - 1 for no phase shift.

Parameter	Descripton
CLKOS3_FPHASE	CLKOS3_FPHASE in the verilog file generated by Diamond Clarity Designer. Choose 0 for no phase shift.

31. ChipScope

The ChipScope Pro system from Xilinx is a tool which provides capabilities for on-chip debugging. ChipScope is being integrated into the target design and thereby provides access to internal signals of the design. It features functionality of a logic analyzer, such as advanced trigger configurations for detection of relevant events and means for displaying the recorded data. The ChipScope system can be added and configured directly through the SpartanMC toolchain.

31.1. System Setup

The ChipScope system is composed of individual modules. This allows for extensive customization for the actual needs. The system is setup by adding, configuring and connecting the ChipScope modules, which are the Integrated Controller (ICON) and the Integrated Logic Analyzer (ILA). Every ILA module must be connected to a dedicated control port of the ICON module. These control ports (*CONTROL0...15*) are the only ports of the ICON module. Each ILA core features one control port (*CONTROL*) for connection with the ICON. Besides, the ILA also has multiple signal inputs for signals to measure. These are called trigger ports (*TRIG0...15*). Additionally, there are ports for a clock input (*CLK*), an optional data input (*DATA*) and an optional trigger output (*TRIG_OUT*).

Note: There must be only one ICON Core in the design. While the SpartanMC toolchain allows creating configurations with more than one ICON Cores, the ChipScope Analyzer software does not support the use of multiple ICONs.

31.2. Module Parameters

31.2.1. Integrated Controller (ICON)

Table 31-124: ICON module parameters

Parameter	Description
NUMBER_CONTROL_PORTS	Number of ILA Cores to be connected.
BOUNDARY_SCAN_CHAIN	BSCAN USER scan chain number to be used.

31.2.2. Integrated Logic Analyzer (ILA)

Table 31-125: ILA module parameters

Parameter	Description
SAMPLE_ON	Defines whether data shall be captured on rising or falling edge of the incoming clock signal.
SAMPLE_DATA_DEPTH	Depth of the data buffer.
ENABLE_STORAGE_QUALIFICATION	Enables the use of optional storage qualifiers via the ChipScope Analyzer.
DATA_SAME_AS_TRIGGER	Defines whether the trigger signals shall be captured or if the data to be captured is supplied through an additional data input port.
DATA_PORT_WIDTH	Defines the width of the optional data input port.
NUMBER_OF_TRIGGER_PORTS	Sets the number of available trigger input ports.
MAX_SEQUENCE_LEVELS	The maximum length for sequencing trigger events in the ChipScope Analyzer. A value of 1 means no sequencing.
ENABLE_TRIGGER_OUTPUT_PORT	Enables an additional 1 bit output port, which is activated whenever the active trigger event is detected.
USE_RPMS	Enables the use of relative-placed macro constraints for performance optimized logic placement of the core. It is recommended to leave option enabled. Disable, if constraints can't be fulfilled (may happen for resource intense designs).
TRIGGER_PORT_WIDTH_0...15	Width of the individual trigger ports.
MATCH_UNITS_AMOUNT_0...15	The amount of match units at the corresponding trigger port. The total amount of match units per ILA core may not exceed 16.
COUNTER_WIDTH_0...15	Adds an optional counter to the corresponding trigger port and sets the counters width.
MATCH_TYPE_0...15	Specifies the type of the match units for the corresponding trigger port
EXCLUDE_FROM_DATA_STORAGE_0...15	Excludes the signal at the corresponding trigger port from being stored in case of a trigger event.

Table 31-126: Types of match units

Type	Bit Values	Match Function
Basic	0, 1, X	'=', '<>'
Basic with edges	0, 1, X, R, F, B, N	'=', '<>'
Extended	0, 1, X	'=', '<>', '<', '>=', '<', '<='
Extended with edges	0, 1, X, R, F, B, N	'=', '<>', '<', '>=', '<', '<='

Type	Bit Values	Match Function
Range	0, 1, X	'=', '<>', '<', '>=', '<', '<=', 'in range', 'not in range'
Range with edges	0, 1, X, R, F, B, N	'=', '<>', '<', '>=', '<', '<=', 'in range', 'not in range'

31.3. Usage

All the debugging, e.g. defining trigger events for data capture and evaluating the recorded signals, is done in the Xilinx ChipScope Analyzer. After adding the ChipScope system to a design, completing the build process and loading the design onto the hardware, the Analyzer can be started by running the following command from the project directory:

make chipscope

To connect the Analyzer with the target design, the used connection method has to be chosen in the *JTAG Chain* menu in the Analyzers user interface. After selecting the adequate option, the Analyzer will do a JTAG scan to detect ChipScope on the hardware. After confirming the following dialogs the automatically created project will be loaded and the system is ready to use.

31.3.1. Bus / Pin Names

When debugging modules using Chipscope, one often ends up adding a new module output port with a number of internal signals, so that they are accessible to Chipscope. To display the correct names of the internal signals, one can add a file called *internal_ports.v* in the module's directory (right next to the XML description file), containing the part of the Verilog file where the internal signals are assigned to the output port.

For example, the following file would assign the name *a* to bit 0 of port *debug* , and the bus *b* to bits 1 and 2:

```
assign debug[0] = a;
assign debug[2:1] = b;
```

32. AXI-Bus-Master

32.1. Overview

The Advanced eXtensible Interface Standard (AXI) is a wide-spread burst based protocol for chip-internal communication in SoC. This DMA module allows to exchange data with AXI slaves. It does not provide any peripheral registers, so all control, status and payload data is transmitted via the DMA memory.
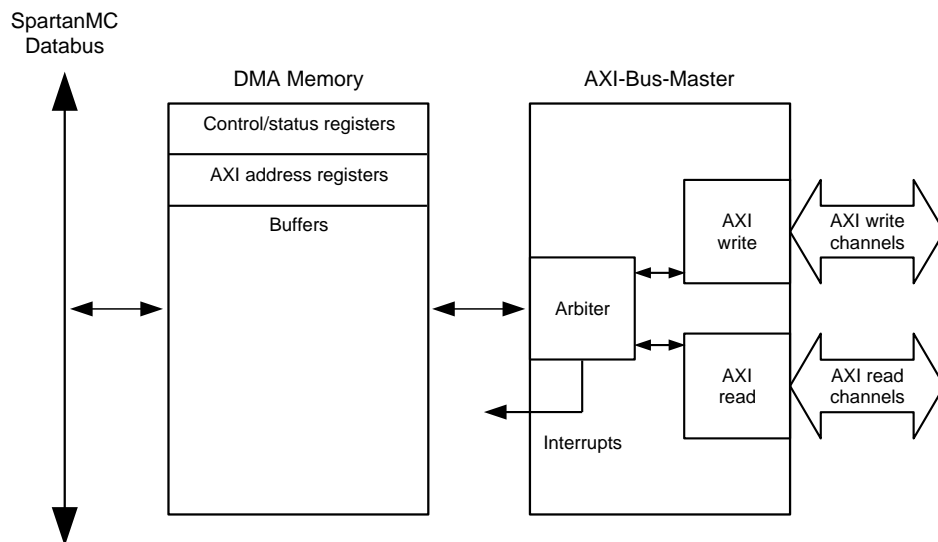
Figure 32-55: AXI-Bus-Master block diagram

32.2. Module parameters

Table 32-127: AXI module parameters

Parameter	Default Value	Description
DMA_BASE_ADR	0x00	Base address of the module's DMA memory. **This parameter is set by jConfig automatically.**
AXI_BUS_WIDTH	16	Width of the AXI read/write data signals. Values of 8, 16 and 32 bit are supported.
DOUBLE_BUFFERING	0	If double buffering is activated, two read and write buffers can be used. Each buffer has its own control and AXI address registers.

Parameter	Default Value	Description
USE_INFER_BRAM	1	Use inferred BRAM instead of macro instantiantion to provide a higher code portability. **When using the inferred BRAM, 8 bit AXI bus width is not supported.**

32.3. DMA Memory Organization

By default, the 16 bit wide DMA memory contains one buffer for read and one buffer for write transactions. Each buffer has its own control/status and AXI address registers. When double buffering is activated, both buffers are divided into two seperate buffers and additional control/status and AXI address registers are enabled. This allows to make better use of the SpartanMC peripheral bus when transmitting big amounts of data. Consider the different maximum burst lengths for single/double buffering.

Offset	Content	Description
0x000	w_ctrl_0	Control and status register of write buffer 0.
0x001	w_ctrl_1	Control and status register of write buffer 1. Only used if double buffering is activated.
0x002	w_ctrl_0	Control and status register of read buffer 0.
0x003	w_ctrl_1	Control and status register of read buffer 1. Only used if double buffering is activated.
0x004	w_addr_0_0	Lower 16 bit of AXI address of write buffer 0.
0x005	w_addr_0_1	Upper 16 bit of AXI address of write buffer 0.
0x006	r_addr_0_0	Lower 16 bit of AXI address of read buffer 0.
0x007	r_addr_0_1	Upper 16 bit of AXI address of read buffer 0.
0x008	w_addr_1_0	Lower 16 bit of AXI address of write buffer 1. Only used if double buffering is activated.
0x009	w_addr_1_1	Upper 16 bit of AXI address of write buffer 1. Only used if double buffering is activated.
0x00A	r_addr_1_0	Lower 16 bit of AXI address of read buffer 1. Only used if double buffering is activated.
0x00B	r_addr_1_1	Upper 16 bit of AXI address of read buffer 1. Only used if double buffering is activated.
0x00C	w_buffer_0	Write buffer 0 with a size of 506 16 bit words. If double buffering is activated, only 252 words can be used.
0x10A	w_buffer_1	Write buffer 1 with a size of 252 16 bit words. Only used if double buffering is activated.
0x206	r_buffer_0	Read buffer 0 with a size of 506 16 bit words. If double buffering is activated, only 252 words can be used.
0x304	r_buffer_1	Read buffer 1 with a size of 252 16 bit words. Only used if double buffering is activated.

Table 32-128: Position of registers and buffers in the DMA memory

Buffering type	8 bit AXI width	16 bit AXI width	32 bit AXI width
single	256	256	253
double	256	252	126

Table 32-129: Maximum burst lengths at different AXI bus widths

32.4. Control Register Organization

Bit	Name	Description
7-0	Burst length	Number of transfers of the AXI transaction.
9-8	Burst type	Burst type of the AXI transaction.
10	valid	Valid bit to initiate the AXI transaction.
12-11	AXI response	The slaves AXI response. Set simultaneously with the done bit.
13	done	Done bit set by the AXI slave when the transaction is finished.
15-14	not used	-

Table 32-130: Control register layout

32.5. Usage

An AXI transaction can be initiated either by setting the AXI address and control registers of the buffer to be used or by using the C-functions that automate the register access for incremental AXI transactions. When transferring big amounts of data, direct register access could reduce control overhead significantly. Besides the "axi_write_sync"/"axi_read_sync" functions, that contain a polling routine, the "axi_write"/"axi_read" functions can be used in combination with the "axi_done" polling function especially to make use of the benefits of double buffering. The module also provides an interrupt signal for each buffer. To reset an interrupt, the done bit in the control/status register needs to be overridden.

Bit	Description
0	Transaction on write buffer 0 finished.
1	Transaction on write buffer 1 finished.
2	Transaction on read buffer 0 finished.
3	Transaction on read buffer 1 finished.

Table 32-131: Interrupt signal structure

32.6. AXI-Bus-Master C-Header for DMA Memory Description

```c
#ifndef __AXI_HW_H
#define __AXI_HW_H

#ifdef __cplusplus
extern "C" {
#endif

//Control-register masks
#define AXI_CTRL_BLEN      0x00FF
#define AXI_CTRL_TYPE      0x0300
#define AXI_CTRL_VALID      0x0400
#define AXI_CTRL_RESP      0x1800
#define AXI_CTRL_DONE      0x2000

//Burst-types
#define AXI_CTRL_TYPE_FIXED   0x0000
#define AXI_CTRL_TYPE_INCR   0x0100
#define AXI_CTRL_TYPE_WRAP   0x0200

typedef struct axi {
    //0x00
    volatile unsigned int   w_ctrl_0;
    //0x01
    volatile unsigned int   w_ctrl_1;
    //0x02
    volatile unsigned int r_ctrl_0;
    //0x03
    volatile unsigned int   r_ctrl_1;

    //0x04
    volatile unsigned int      w_addr_0_0;
    //0x05
    volatile unsigned int      w_addr_0_1;
    //0x06
    volatile unsigned int       r_addr_0_0;
    //0x07
    volatile unsigned int       r_addr_0_1;

    //0x08
    volatile unsigned int      w_addr_1_0;
    //0x09
    volatile unsigned int      w_addr_1_1;
    //0x0A
```

```
    volatile unsigned int       r_addr_1_0;
    //0x0B
    volatile unsigned int       r_addr_1_1;


    //0x0C
    volatile unsigned int               w_buffer_0[254];
    //0x10a
    volatile unsigned int               w_buffer_1[252];
    //0x206
    volatile unsigned int   r_buffer_0[254];
    //0x304
    volatile unsigned int   r_buffer_1[252];
} axi_dma_t;

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/axi.h

33. Global Firmware Memory

33.1. Overview

The global firmware memory gives multiple SpartanMC cores access to the same Block-RAM-based memory. It stores one firmware, which is executed by every connected core and has to be added as a common module. The global memory module, generates one code memory and for each attached core additionally one data memory.

33.2. Module parameters

Table 33-132: Global firmware memory module parameters

Parameter	Default Value	Description
CORES_COUNT	2	Number of cores to connect
RAMBLOCKS_GLOBAL	4	Ramblocks used for global memory
USE_TWO_PORTS_GLOBAL	0	Enable the use of both ports of the internal block RAMs. Can be faster when the memory is heavily used, but needs more resources.
LOCAL_BASE_ADDR	8192	Base address of the local data memories.
RAMBLOCKS_LOCAL	4	Ramblocks used for local memory
SHOW_MEM_ALL	0	Print all memory accesses during simulation
SHOW_MEM_SINGLE	1	Have one memory print accesses during simulation
CACHE_SIZE	1024	The amount of cache Blocks to be used
OFFSET_LENGTH	1	Length of the caches block offset in bits
CACHE_WAYS	1	N-way set associative cache
INDEX_LENGTH	9	Number of index bits for the cahce.

33.3. Restrictions for connected subsystems

All connected subsystems must use the same firmware. It is not possible to use individual firmware parts, since the instruction fetch from the local memory is entirely disabled.

The subsystems must also have the same memory layout and the same peripherals. Otherwise they would need different code.

34. Router for multicore systems

The router implements a FIFO through which multiple SpartanMC cores are able to communicate. Routers can send, receive and pass messages. Static routing is used. Also the route has to be established from sender to receiver until the first data is transmitted.

34.1. Requirements

Every subsystem should contain only one routermodule. The routerids should be integers starting with 0. The Buffers should contain at least 4 values. The submodule "sender" is only synthesized if the router has at least one output. The submodule "receiver" is only synthesized if the router has at least one input. So the input "0" from the submodule "selector" is connected to the submodule "sender" or the first input. So the output "0" from the submodule "splitter" is connected to the submodule "receiver" or the first output. The TO_DEST_x parameters in jconfig describe to which port the splitter is routing (x is a destination routerid). If the router has a receiver "0" means route to own receiver and "1" to first output port. If the router has no receiver "0" means route to first output port. The maximum allowed messagesize is "buffersize-1". If the firmware at the receiver is not reading the buffer and it filled at a level that there is not enough space for new messages, the messages will not be accepted (returnout=01). There are three signals between two connected routers: data, request and return. If router A wants to send to router B then data (18 bits) and request (1 bit) is connected from A to B , but return (2 bits) is connected from B to A. Every SpartanMC-Core and routermodule has to be driven by the same clock.

Output (Value of TO_DEST_x)	meaning if router has a receiver	meaning if router has no receiver
0	to own receiver	to first output port
1	to first output port	to second output port
2	to second output port	to third output port
3	to third receiver	to fourth output port

Table 34-133: Outputs of splitter (TO_DEST_x)

It is required, that the first 18 bits of datain, the first 1 bit of requestin and the first 2 bits of returnout are connected to the same router and so on.

Routernumber	datain	requestin	returnout
first input	[17:0]	[0]	[1:0]

Routernumber	datain	requestin	returnout
second input	[35:18]	[1]	[3:2]
third input	[53:36]	[2]	[5:4]
fourth input	[71:54]	[3]	[7:6]

Table 34-134: Input bits

It is required, that the first 18 bits of dataout, the first 1 bit of requestout and the first 2 bits of returnin are connected to the same router and so on.

Routernumber	datain	requestin	returnout
first output	[17:0]	[0]	[1:0]
second output	[35:18]	[1]	[3:2]
third output	[53:36]	[2]	[5:4]
fourth output	[71:54]	[3]	[7:6]

Table 34-135: Output bits

34.2. Module Parameters

Parameter	Default Value	Descripton
ROUTER_ID	0	The id of the rounter. Should be x if the router is part of subsystem_x. (0,1,2,..)
DEPTH_OUTPUT_BUFFER	32	Amount of 18bit Values stored in the sender fifo. Should be greater or equal 4.
DEPTH_INPUT_BUFFER	32	Amount of 18bit Values stored in the receiver fifo. Should be greater or equal 4.
AMOUNT_OF_INPUTS	0	Amount of inputs (from other routers)
AMOUNT_OF_OUTPUTS	0	Amount of outputs (to other routers)
TO_DEST_0	0	Which output should the splitter use for messages to router 0.
TO_DEST_1	0	Which output should the splitter use for messages to router 1.
TO_DEST_2	0	Which output should the splitter use for messages to router 2.

Table 34-136: Module parameters

34.3. Java routing tool

There is a java tool reading your jconfig.xml, using Dijkstra's algorithm, creating a .dot file for graph creation using graphviz and telling you what to insert in jconfig in the TO_DEST_x fields.

To use this tool some naming conventions have to be satisfied. Every output a router has has to be connected to a net (Add gluelogic -> internal -> net) in the same subsystem. This net needs to have a specific name. Every dataout and requestout net has to be named net_dataout_x and net_requestout_x, where x is the number of the splitter output the net is connected to (0,1,2,... if the router has no receiver, 1,2,3,... if the router has a receiver). Every returnout net has to be named net_returnout_y, where y is the number of the selector output the net is connected to(0,1,2,... if the router has no sender, 1,2,3,... if the router has a sender).

To use the tool go to the project rootfolder. There the jconfig file should be named "jconfig.xml". Enter "make routing" in the terminal. There will be 4 files created.

File	Content
routingtable_debugoutput.txt	Contains what the tool has detected and errors (if any)
routingtable_FromViaTo.txt	Contains what connections the tool has detected. Notation: from_routerid:splitter_output_port:to_routerid
routingtable_graph.dot	Use "neato -Tsvg routingtable_graph.dot -o routingtable_graph.svg -Gstart=rand" to create a svg graph.
routingtable_text.txt	The output of Dijkstra's algorithm. This is what you should enter in jconfig. All fields not listed in this file should be "0". "0" has several meanings: not reachable, to own receiver(if the router has one), or to first output port(if therouter has no receiver). In fact packages for not reachable destinations will be forwarded. You are responsible that this will never happen.

Table 34-137: output from 'make routing'

34.4. Developer information

see $SPARTANMC_ROOT/src/doc/users-manual/src/router/*

Return wire	Meaning
00	Default. Receiver, or a transport router on the path has not accepted or denied yet.
01	Denied. Either a transport router, or the receiving router is busy. Or maybe the buffer od the receiving router has not enough free entries.
10	Accepted. Every transport router and the receiving router can handle the request. A dedicated line is established from submodule sender to submodule receiver. Receiver buffer has enough free entries. Ready for transfer. This state will be keept until the transmission is finished.
11	Finish. This signal is sended one clock period after sending is completed (state=10, request switches from 1 to 0). After sending one clock cycle, signal will switch to 00 again.

Table 34-138: meanings of return bits

34.5. Peripheral Registers

The router modules have three registers each for message transfer.

34.5.1. Router C-Header for Register description

```
#ifndef __router_H
#define __router_H

#ifdef __cplusplus
extern "C" {
#endif

typedef struct {
    volatile unsigned int data;
    volatile unsigned int free_entries;
    volatile unsigned int data_available;
} router_regs_t;

//router->data (router_start_addr + 0)
//router->free_entries (router_start_addr + 1)
//router->data_available (router_start_addr + 2)

#ifdef __cplusplus
}
#endif

#endif
```
Code included from spartanmc/include/peripherals/router.h

34.5.2. data Register Description

The data register can be written for sending, or read for receiving messages.

34.5.3. free_entries Register Description

The free_entries register is only used for sending messages. It contains how many buffer entries are free. A new message should only be transfered to the sender's buffer via (data register) if there are enough free buffer entries to store the whole message including the header. (router->free_entries >= messagesize+1)

34.5.4. data_available Register Description

The data_available register is only used for receiving messages. it contains how many buffer entries are used. If it is greater than zero, a new header package is stored in the buffer (because new messages always start with a header). It can be read an now router_read.c can extract the messagesize from the header. The message is only transmitted, if the whole message is in the receiver's buffer. (router->data_available >=1) (router->data_available >= messagesize)

34.6. Usage examples

There are the wrappers router_check_data_available.c, router_read.c and router_send_data.c. Only the use of this wrappers is documented here.

34.6.1. router_check_data_available

```c
#include <system/peripherals.h>
#include <router.h>

void main() {
   unsigned int dataavailable;
   dataavailable = router_check_data_available(ROUTER_0);
   while(1);
}
```

34.6.2. router_read

```c
#include <system/peripherals.h>
#include <router.h>

void main() {
   unsigned int data[10] = {3,3,3,3,3,3,3,3,3,3};
   unsigned int msgsize, source;
   router_read(ROUTER_0, &data, &msgsize, &source);
   while(1);
}
```

34.6.3. router_send_data

```c
#include <system/peripherals.h>
#include <router.h>

void main() {
   //router_send_data (router, data, source, msgsize, dest)
   unsigned int data[10] = {193,1,2,3,4,5,6,7,8,9};
   router_send_data (ROUTER_0, &data, 10, 10, 5);
   while(1);
}
```

35. DVI output

The DVI peripheral allows outputting video data to an attached Chrontel CH7301C (as is present on the SP605 eval board). Because the memory needed to store the images is very large, Block RAMs would be too small. Therefore, Data is sourced from an external DRAM, using the ddr_mcb_sp6 module.

35.1. Module Parameters

Most parameters are concerned with the size of the visible area and the blanking intervals. The values needed are dependent on the connected monitor. The monitor's supported resolutions and blanking intervals can be read via a separate I2C connection. An overview of the parameters can be seen in the following figure. Note that the order of back porch and front porch may seem to be swapped (e.g. *front* porch comes *after* visible area). This is not the case, because they are named in relation to the Sync Interval.
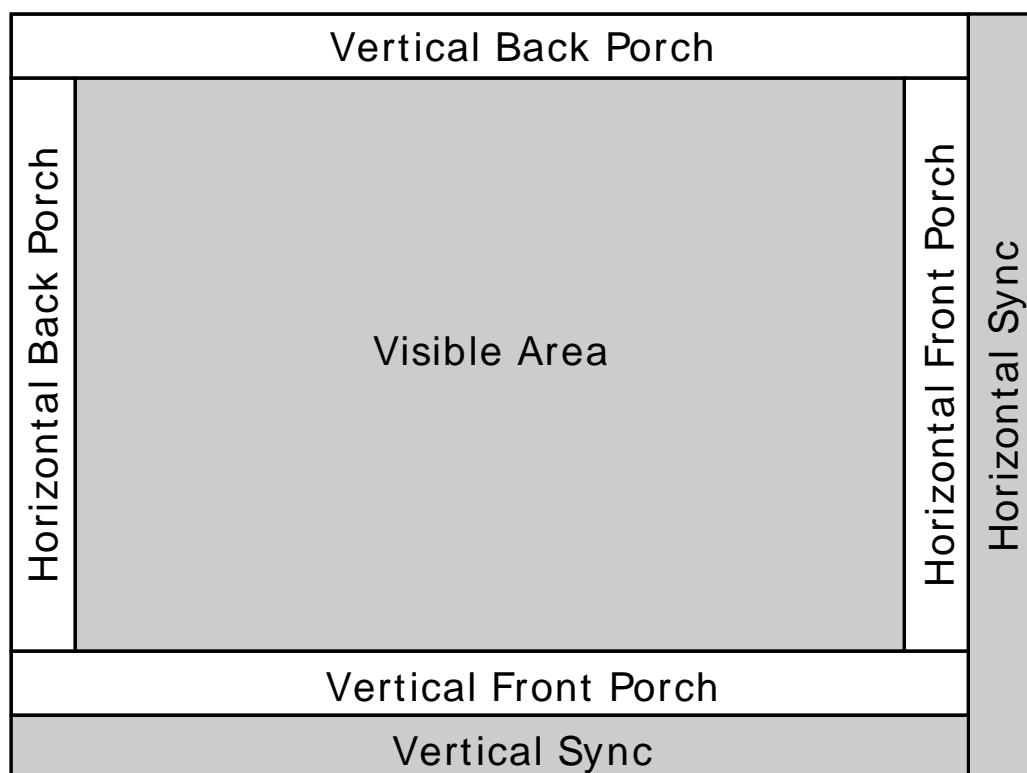
Figure 35-56: Sync intervals

The total number of pixels/lines output in horizontal/vertical directions is the sum of back porch, visible area, front porch and sync. The total number of pixels per frame is the product of the total number of horizontal pixels by the total number of vertical lines. The number of frames per second is the the frequency of the pixel clock divided by the total number of pixels per frame.

Normally, both the horizontal sync and vertical sync signals are active high. However, some monitors support reduced blanking time modes, where one or both of these lines may need to be inverted.

Parameter	Default Value	Descripton
H_VISIBLE	1280	Horizontal visible pixels
H_FRONT_PORCH	48	Duration of horizontal front porch
H_SYNC	112	Duration of horizontal sync
H_BACK_PORCH	248	Duration of horizontal back porch
V_VISIBLE	1024	Vertial visible pixels
V_FRONT_PORCH	1	Duration of vertical front porch
V_SYNC	3	Duration of vertical sync
V_BACK_PORCH	38	Duration of vertical back porch
H_SYNC_INVERT	0	Invert horizontal sync flag
V_SYNC_INVERT	0	Invert vertical sync flag
COLOR_MODE	RGB_24	The color mode to use. See memory layout for details

Table 35-139: Module Parameters

35.2. Peripheral Registers

35.2.1. Enable Register Description

The enable signal can enable or disable the output.

Table 35-140: Enable register

Offset	Name	Access	Description
0	Enable	read/ write	Enable video output

35.3. Memory Layout

Only a single frame can be read from memory, which always has to start at address zero. Therefore, it is not possible to use double buffering. The pixels are arranged row by row from top to bottom. Within the row, they are ordered from left to right. Only the visible area is stored.

Note: The select memory layout has to be set in the Chrontel chip via the dedicated I2C connection as well.

Depending on COLOR_MODE, pixels either occupy two or four bytes:

35.3.1. RGB Color Mode

Every pixel consists of 32 Bits, of which 8 are unused. The remaining 24 Bits provide 8 Bit per color channel.

Table 35-141: Pixel Data in RGB mode

Offset	Name
7-0	Blue
15-8	Green
23-16	Red
31-24	Unused

This corresponds to the chip's input data format 0.

35.3.2. YCRCB Color Mode

Every pixel consists of 16 Bits. Every pixel contains luminance information (Y), while color is the same for two pixels. Even pixels contain the Blue color (Cb), while odd ones contain the red color (Cr).

Table 35-142: Pixel Data in YCrCb mode

Offset	Name
7-0	Luminance (Y)
15-8	Color (Cb for even pixels, Cr for odd)

This corresponds to the chip's input data format 4.

36. Ethernet

The ethernet controller is split into three modules: MDIO, Ethernet TX and Ethernet RX. Different modules may be connected to different SpartanMC cores.

Currently, only MII is supported. GMII is downward compatible to MII. If the MDIO registers are set up so that only 100 MBit connections are negotiated, this controller can be used. RGMII and RMII have to be adapted to GMII/MII externally.

36.1. MDIO

MDIO is the Ethernet standard's Management interface. It is used to set configuration registers. The bus can address up to 32 Ethernet Phys. There are only 32 registers addressable per Phy. Current Phys provide many more than that. To access those, indirect addressing schemes have to be utilized.

MDIO consists of a clock line and a bidirectional data line. On some FPGA architectures, one bidirectional signal cannot be fed into multiple FPGA pins. Since different phys may be connected to different pins, a WIDTH parameter is provided to generate multiple connections.

36.1.1. Module parameters

Table 36-143: MDIO module parameters

Parameter	Default Value	Description
Width	1	Number of MDIO data lines

36.1.2. Module Registers

Table 36-144: MDIO registers

Offset	Name	Access	Description
0	MDIO Data	read/ write	Contains the Data
1	MDIO Address	read/ write	Contains both Phy and Register address and the status.

Writing to the MDIO data register starts a transmission. This causes the ready flag in the MDIO address register to go low until the transmission is finished.

As soon as the memory location pointed at by the DMA data offset register is not zero (either caused by the memory address being written to, or the register being changed), the frame is transmitted. Therefore, the length may only be written after the frame contents have been written.

After transmitting a packet, the hardware updates the offset register to point to the next address after the frame. To avoid sending an unintended packet, this memory location should be initialized to zero before writing the original packet's length.

36.2.2. Module Registers

Table 36-147: Ethernet TX registers

Offset	Name	Access	Description
0	Status/Control	read/write	Status flags
1	DMA data offset	read/write	Contains the current data pointer into the DMA memory
2	Interrupt	read/write	Interrupt signal
3	Packet count	read	Number of packets sent since reset

36.2.3. Status/Control Register

Table 36-148: Ethernet TX status/control register layout

Bit	Name	Access	Default	Description
0	IrqEn	read/write	0	Interrupt enable
1	Tx Available	read	0	A frame is available to be transmitted
2	Tx Transmitting	read	0	A frame is currently being transmitted
3-17	unused	read	0	

Table 36-148: Ethernet TX status/control register layout

36.2.4. DMA data offset

Table 36-149: Ethernet TX DMA offset register layout

Bit	Name	Access	Default	Description
0-9	DMA data offset	read/ write	0	The offset within the DMA memory that the next packet is at. This is incremented by the hardware if a frame has been transmitted
10-17	unused	read	0	

Table 36-149: Ethernet TX DMA offset register layout

36.2.5. Interrupt Register

Table 36-150: Ethernet TX Interrupt register layout

Bit	Name	Access	Default	Description
0	Register Address	read/ write	0	Interrupt flag. Write to acknowledge interrupt.
1-17	unused	read	0	

Table 36-150: Ethernet TX Interrupt register layout

36.2.6. Packet count Register

Table 36-151: Ethernet TX packet count register layout

Bit	Name	Access	Default	Description
0-17	Packet count	read	0	The number of packets that have been transmitted since reset

Table 36-151: Ethernet TX packet count register layout

36.3. Ethernet RX

This module handles the RX part of the MII interface

36.3.1. DMA memory

Memory is organized as a ring buffer. Every entry starts with the length in bytes, stored as an 18 Bit int, followed by an Ethernet frame. The Frame check sum (FCS) is part of the received frame and visible to the software. The address of the oldest package currently in memory is stored in the DMA offset register.

The hardware takes care that the length field is zeroed before a valid frame has been received. After a frame has been fully received and the CRC checked, its length is prepended to the data in the memory.

When a package is not needed any more, it must be explicitly discarded. This is done by setting the offset register to the first address after the frame. If the memory is full, all subsequent packages are dropped, until the software discards enough packages in memory.

Note: The hardware writes to the memory in bytes. The frame's length is therefore written as two half-words in two consecutive cycles. Polling code like this

```
int bytes = ETHERNET_RX_DMA.x[ETHERNET_RX.offset];
if (bytes) {
/* Process a frame of length <bytes> */
}
```

is incorrect! If the hardware has written one of the length's half words in the previous cycle and is writing the second half word while the memory is being read, a **wrong value** will be read as the length. When discarding the packet, the offset will then be set to an incorrect value that probably does not line up with the start of a new frame, and frame data will be interpreted as the length field. Only a reset can recover from this error.

Instead, first read the value and check if it is zero, then read it again and use that as the length of the frame:

```
if (ETHERNET_RX_DMA.x[ETHERNET_RX.offset]) {
int bytes = ETHERNET_RX_DMA.x[ETHERNET_RX.offset];
/* Process a frame of length <bytes> */
}
```

The interrupt flag is only raised after both parts of the length have been written, so it is not necessary to read the length twice when using interrupts.

Note: Frames may start near the end of the DMA memory and continue at the start of the memory. Care must be taken to read at the correct addresses. It is best to use the provided functions.

36.3.2. Module parameters

Table 36-152: Ethernet RX module parameters

Parameter	Default Value	Description
MAC_ADDRESS_HIGH	0x0010A4	Top three byte of the MAC address
MAC_ADDRESS_LOW	0x7BEA80	Lower three byte of the MAC address

36.3.3. Module Registers

Table 36-153: Ethernet RX registers

Offset	Name	Access	Description
0	Control	read/ write	Control flags
1	DMA data offset	read/ write	Contains the current data pointer into the DMA memory
2	Interrupt	read/ write	Interrupt signal
3	Packet count	read	Number of packets received since reset
4	CRC error count	read	Number of packets with incorrect CRC sums received since reset
5	MAC Register 1	read	Low 16 Bit of MAC. Provides access to the configured value.
6	MAC Register 2	read	Middle 16 Bit of MAC. Provides access to the configured value.
7	MAC Register 3	read	High 16 Bit of MAC. Provides access to the configured value.

36.3.4. Control Register

Table 36-154: Ethernet RX control register layout

Bit	Name	Access	Default	Description
0	Promiscuous mode	read/ write	0	Receive every frame, regardless of destination MAC. Takes precedence over Ignore Broadcast
1	IrqEn	read/ write	0	Interrupt enable
2	Ignore Broadcase	read/ write	0	Do not receive broadcast frames

Bit	Name	Access	Default	Description
3	Ignore CRC errors	read/write	0	Also receive frames with invalid frame check sum. Independent of MAC address matching
4-17	unused	read	0	

Table 36-154: Ethernet RX control register layout

36.3.5. DMA data offset

Table 36-155: Ethernet RX DMA offset register layout

Bit	Name	Access	Default	Description
0-9	DMA data offset	read/write	0	The offset within the DMA memory that the next packet is at. This is incremented by the hardware if a frame has been transmitted
10-17	unused	read	0	

Table 36-155: Ethernet RX DMA offset register layout

36.3.6. Interrupt Register

Table 36-156: Ethernet RX interrupt register layout

Bit	Name	Access	Default	Description
0	Register Address	read/write	0	Interrupt flag. Write to acknowledge interrupt.
1-17	unused	read	0	

Table 36-156: Ethernet RX interrupt register layout

36.3.7. Packet count Register

Table 36-157: Ethernet RX packet count register layout

Bit	Name	Access	Default	Description
0-17	Packet count	read	0	The number of packets that have been transmitted since reset

Table 36-157: Ethernet RX packet count register layout

37. Simulation using ModelSim

SpartanMC projects can be easily simulated using ModelSim.

37.1. Creating a simulation directory

Inside the project, run `make newsim +path=<path>` to create a simulation directory in the specified subdirectory. A testbench and a start script are automatically generated

37.2. Customizing the simulation

The generated file *testbench.v* is the test bench. By default, it instantiates the toplevel module, connects all its inputs and outputs to regs / wires as appropriate and generates clocks. In most cases, you need to edit it to generate input signals like reset or outputs from peripherals.

In *testbench.fdo* , you can customize the simulation run time and other variables.

37.3. Starting ModelSim

Inside the simulation directory, you can start ModelSim by running `vsim -do testbench.fdo` . Note that the firmware gets compiled automatically.

38. JConfig 4

38.1. Overview

JConfig is the core utility that allows to view and configure a spartanmc project. It is implemented in Java and can be found in */src/javaTools* . JConfig has its own XML-based config-file format (currently .jc4) and generates the entirety of the *system/* directory in a SpartanMC project, including the top-level HDL, header files for the firmware and several types of config files to configure make, initramj, the synthesis and simulation tools and compilers.

38.2. Implementation Structure

JConfig uses libdevxml to read and organize all the descriptions of target boards, devices and modules that it can use to build a project. These descriptions can be found in */spartanmc/hardware* , */spartanmc/buses* , */spartanmc/devices* and */spartanmc/targets*

The core functionality and logic of JConfig can be found in *libjconfig* , of which *libjconfig-api* is the API that should be used in all other dependents to keep everything modular.

The *jconfig* module only includes the Java Swing UI, and is not needed to open, modify or analyze a JConfig configuration.

Other libraries that start with a libjconfig- prefix are modular extensions to libjconfig which are already dependants of libjconfig. Examples of this are libjconfig-import which is capable of reading and importing a Microstreams hardware.xml into libjconfig, libjconfig-script-groovy, which adds support for the commandline-only groovy-shell, the groovy-shell available from the UI and the automatic execution of the groovy-scripts that come with module-descriptions.

38.3. Documentation

More details to each part of libjconfig can be found in JavaDoc, as each library should have its overview going into more details about its structure and content

39. JConfig 4 Scripting

libjconfig includes groovy-scripting support. Especially in JC4 this means mainly to operate programmatically on the libjconfig-api-provided document-model. The Java-API was overhauled significantly to make properties and modifications much more obvious and accessible, nevertheless there are more convenient methods, properties and operators available in the groovy-language (which are either standard-groovy or defined in libjconfig-api-groovy-extensions, mostly as extension methods)

Groovy allows various simplifications of the Java-Syntax without changing its meaning. It's almost always okay to write standard Java-code in groovy, most groovy constructs are just shorthand for things that already exist in Java (with the exception of lambda-functions, which are slightly different and more powerful in groovy). Here a few examples:

- Parentheses of methods can be eschewed: `add a,b` is equivalent to add(a, b)

- Semicolons are optional

- There are list and map literals: `["bla"]` is an ArrayList<String>() with the entry "bla", `[key:"value"]` is a LinkedHashMap<String, String>()

- `def` is a stand-in for any available type (similar to "var" in Java 11)

- JavaBean Properties (proper getters and setters in java) can be used as properties in groovy: `obj.property = 5` is equivalent to `obj.setProperty(5)`, same applies to getters

- Groovy is dynamic, in that it automatically will cast to a type on which the operation is available and it will access private Java methods and fields if nothing else of the same name exists

- Groovy supports operator-overloading. The []-operator knwon from Java-Arrays is available on lists, and maps

- Groovy can fake members on-the-fly. On Maps and similar structures, one can access map-entries (with String-keys) just as properties: `map.keyName == map["keyName"]`, both of which are equivalent to Javas `map.get("keyName")`

- Groovy can inline variables in strings: "This is a string with $var", var will be resolved similar to string-concatenation in Java

When opening the javaTools project in IntelliJ IDEA, IntelliJ makes the entire libjconfig-api available from groovy, autocompletes and type-checks groovy scripts when possible. Underlined groovy-code in IntelliJ indicates that this method/property is not available statically or publicly and if one does not know what one is doing leads to highly unstable code. Yellowish marked code indicates that IntelliJs type-checker cannot see how this is valid, but groovy may or may not make it work at runtime. This means good code that is not just one-off, should be statically typed, and fix all of IntelliJ's warnings.

39.1. Shell-Scripting

This is designed to simplify and automate various tasks on the users behest. I.E. one can batch-transform a document or change multiple parameters in a coordinated, yet automated way or run additional checks for ones project or just look at certain internal properties that are currently not exposed through the normal UI. This should be considered low-impact and is perfectly fine to be used if the user prefers a keyboard-centric interaction with jconfig. There is a commandline-only version of this that comes up if jconfig gets started with "--quiet" and no other actions are commanded, but this version is rarely used and might be missing some of the helper functions that are available from within the UI for internal reasons (auto-completion is better though).

39.2. Event-Based scripting

This is a much more dangerous way of scripting if one is not very familiar with the internals and the document-model. It should be considered a last-resort. When scripts are used to add valuable functionality that is not otherwise available, this should be considered temporary. Longterm one should always strive to add that functionality directly to libjconfig, as it is much easier to integrate, manage and maintain there, especially if the functionality is rather complicated. In the past we had many scripts that unknowingly lead to endless recursions, document corruption or made jconfig unstable in certain corner-cases.

libjconfig looks for scripts with the following pattern: [moduleType-Name].[eventType].groovy. These scripts can either be located directly next to their module-descriptions in the respective source-tree or directly in the project's folder. Project-level scripts replace more widely available ones if they exist. ModuleTypeName is the name of the module.xml and eventType is one of:

- isValid: the script is expected to return a boolean to indicate passing or failing additional validity checks (per instance), the boolean variable "report" is passed into the script to indicate whether the script should create issues in response to problems or not

- onAdd: script may do smart initialization of a newly added module (configure default values that depend on other elements in the document). Or add additional modules. Although changing other modules is technically possible it is not expected by the user and should be avoided

A document-wide isValid is possible with the name "document.isValid.groovy". The document will also be passed in the *module* variable.

libjconfig looks for scripts in all hardware-source-trees (by default the /spartanmc/hardware folder and directly in the current project-folder) that match the following pattern: [eventType].groovy. These scripts are executed once per document. Project-level scripts replace more widely available ones if they exist. EventType can be one of:

- onSaving: called directly before saving allows last-minute cleanups

- onCreatedDocument: called after a new document was created. Only intended for project-specific
- onExportingDocument: called before exporting a document
- onImportedDocument: called after importing a document

39.3. Available Properties and Methods for scripting

All scripts will have the following read-only variables available to them:

- `document` (also available as `doc`): reference to the document
- `module`: the current module (does not exist if the script does not run in the context of a module)
- `project`: reference to the current project
- `library`: reference to the library of available module and targets
- `documentHandler` (also available as `docHandler`): reference to the documentHandler that allows opening, saving, moving and creating documents

All scripts will have the following "global functions" available to them (does not work on commandline-shell):

- `autoLayoutAll()`: document.autoLayoutAll(), autoLayouts all addressLayouts
- `build(boolean all = false)`
- `save(String file = null)`
- `newDocument(String targetString)`
- `open(String file)`
- `execute(String file)`: execute the file as groovy-script
- `IPinHandler pin(String name)`: lookup name as pinHandler (either with leading # or without)
- `StringTreeBuffer createTreeBuffer()`: get a new treeBuffer for elaborate() calls
- `IModuleConfiguration find(String name)`: find a module as long as the name is unique
- `Set<IModuleConfiguration>findByName(String name)`: find all modules with that name
- `Set<IModuleConfiguration>findByType(String name)`: find all modules with that type
- `add(String typeName, String name = null)`: shorthand for document.add(typeName, name)
- `remove(String name): find(name).remove()`

- `withType(String typeName, Closure forEach)` : apply closure to all mathing modules with module as delegate of closure

- `connect(IPortHandler a, IPortHandler b)` : connect to ports

- `connect(IPortHandler a, int aStart, IPortHandler b, int bStart, int width)` : connect partially, use -1 for "AUTO"

- `connect(IBusHandler a, IBusHandler b)` : connect to busConnectors

- `connectIfConnectable(IPortHandler a, IPortHandler b)` : connect to ports, do not complain if not possible

- `connectIfConnectable(IBusHandler a, IBusHandler b)` : connect to busConnectors, do not complain if not possible

- `connectMultiple(IPortHandler a, int aStart = 0, int width = -1, IModuleConfiguration other, List<String> portNames)` : connect partially bit-by-bit

- `connectMultiple(IPortHandler a, int aStart = 0, int width = -1, IModuleConfiguration other, Closure getPort)` : connect partially bit-by-bit, gives closure the current index and expects a PortHandler's name

Many of the functions are simply shorthand for methods on either documentHandler, document.

In addition to the above mentioned functions/methods additional shortcuts are available on modules (document is a module):

- `module.add(String typeName, String name = null)` : add typeName as child, use name or generate unique name

- `module + "typeName"` : add typeName as child, generate unique name

- `module.each { }` : call closure on all children (recursive, depth-first)

- `(module as IModuleFacet)` : shortHand for IModuleFacet.require(module) from Java

- `module.remove()` : remove from parent

- `module.addBusBreakout(String busDefName, Role role, String moduleName = null)`

- `module.addressLayout` : get the addressLayout if module is a Subsystem

- `module.addresses` : get the addressBlocks if module is a peripheral / memory (careful, addresses are local addresses)

- `module[childName]`

- `module.childName`

- `module.ports , document.pins , module.buses , module.connectors` : get the IConnectorSource, which operates similar to a groovy Map with ["connector"] and .connector, the buses / ports variants filter out not matching ones when iterating and cast

- `module.attributes` , `module.parameters` : same for parameteres / attributes

Properties such add addresses, addressLayout, ports etc, should all be automatically printed in some form to the shell in a readable fashion. The contents are usally accessible just like on a groovy map. If the default shellOutput is not sufficient there often is a elaborate() method with more details or everything resolvable such as `spartanmc_0.parameters.FREQUENCY` should automatically explain how it is calculated or also possess a elaborate() method.

39.4. Examples

39.4.1. Script adding and connecting modules

```
add "spartamc" // add spartanmc to document
document.spartanmc.add "uart_light" // add uart_light modules
to subsystem spartanmc
add "clk_xilinx"
clk_xilinx.parameters.RESET_LEVEL.value = "HIGH_ACTIVE"
connect clk_xilinx.ports.clkfx, spartanmc.ports.clk
connect document.pins.CLOCK_USER, clk_xilinx.ports.clkgen
spartanmc.add "port_in"
port_in.parameters.COUNT.value = 4
connectMultiple port_in.ports.pin, doc, ["GPIO_BUTTON_0",
GPIO_BUTTON_1]
port_in.ports.pin.disconnectAll()
connectMultiple port_in.ports.pin, doc, {i -> "GPIO_BUTTON_$i"
}
```

39.4.2. Shell interaction to manually configure addresses

```
spartanmc_0.addressLayout
Script returned: AddressLayout of spartanmc_0
|----AddressSpace VonNeumann
| \----/spartanmc_0/spartanmc_mem_local_0#BASE_ADDR: [[Not
Layouted]] (0x4000 words, configurable)
|----AddressSpace data
| |----/spartanmc_0/intctrl_0#BASE_ADR: [[Not Layouted]] (0x8
words, SubRanged to 0x1fc00, priviliged RW, configurable)
| |----/spartanmc_0/memguard_0#BASE_ADR: [[Not Layouted]] (0x8
words, SubRanged to 0x1fc00, priviliged RW, configurable)
| |----/spartanmc_0/sim_print_0#BASE_ADR: [[Not Layouted]]
(0x2 words, SubRanged to 0x1fc00, priviliged RW, configurable)
```

```
| |----/spartanmc_0/sim_stopper#BASE_ADR: [[Not Layouted]]
(0x1 words, SubRanged to 0x1fc00, priviliged RW, configurable)
| \----/spartanmc_0/uart_light_0#BASE_ADR: [[Not Layouted]]
(0x3 words, SubRanged to 0x1fc00, priviliged RW, configurable)
\----AddressSpace code
sim_stopper.addresses["BASE_ADR"].manualAddress = 0x0
spartanmc_0.addressLayout.autoLayout()
Script returned: AddressLayout of spartanmc_0
|----AddressSpace VonNeumann
| \----/spartanmc_0/spartanmc_mem_local_0#BASE_ADDR: 0x0
(0x4000 words, autoConfigured)
|----AddressSpace data
| |----/spartanmc_0/sim_stopper#BASE_ADR: 0x1fc00 (0x1 words,
SubRanged to 0x1fc00, priviliged RW, persistent)
| |----/spartanmc_0/intctrl_0#BASE_ADR: 0x1fc08 (0x8 words,
SubRanged to 0x1fc00, priviliged RW, autoConfigured)
| |----/spartanmc_0/memguard_0#BASE_ADR: 0x1fc10 (0x8 words,
SubRanged to 0x1fc00, priviliged RW, autoConfigured)
| |----/spartanmc_0/uart_light_0#BASE_ADR: 0x1fc18 (0x3 words,
SubRanged to 0x1fc00, priviliged RW, autoConfigured)
| \----/spartanmc_0/sim_print_0#BASE_ADR: 0x1fc20 (0x2 words,
SubRanged to 0x1fc00, priviliged RW, autoConfigured)
\----AddressSpace code
```

Note that after assigning a manualAddress to sim_stopper, it is described as "persistent", while all other modules are placed around it and are marked "autoConfigured"

MANPAGE – SPARTANMC(7)

NAME

spartanmc – Toolkit for easy implementation of custom SoCs (System-on-Chip) on Xilinx FPGAs

SYNOPSIS

Global targets:

make cablesetup[**+group**=*GROUP*]

make integrity_check +target=*PLATFORM*

make man[*MANPAGE*]

make newproject +path=*PATH*

make reconfigure

make setup

make unconfigure

make[**what**]

Project targets:

DESCRIPTION

The SpartanMC-SoC-Kit is a set of tools for implementing FPGA-based SoCs. The implementation process does not require knowledge about hardware description languages such as Verilog or VHDL.

Based on the 18-bit SpartanMC microprocessor core the SoC-Kit provides a toolchain to allow easy implementation of a custom System-on-Chip (SoC) on a Xilinx FPGA. The frontend used is GNU *make* to invoke a number of underlying backend tools.

To compose an SoC, the GUI-based system builder *jConfig* will generate a set of hardware source code and configuration files based on the users configuration choices. The SoC then is synthesized from this set of files by invoking the corresponding tools from Xilinx ISE Suite. The whole process is driven by *make* which finally generates a bitfile that can be downloaded to your target FPGA.

The configuration also includes system firmware which is written in C and will be embedded into the design during synthesis. An optional bootloader allows for later update of the software components without re-synthesizing the hardware.

MAKE TARGETS

All steps to design a SoC are triggered by *make*. This section describes all available operations implemented as make targets.

Global targets

Global targets are available from the installation directory of the SpartanMC-SoC-Kit. This directory is called *SPARTANMC_ROOT* .

cablesetup   Creates a rules file (*.rules) understood by udev to ensure proper operation of the Xilinx programming tool *impact*. The rules will make sure our system loads the correct USB-firmware to enable the cable driver to detect your cable. If your system has restricted USB access, the option *GROUP* specifies which user group is granted access to the USB programming cable. If ommitted, the default group 'xilinx' is used. After running this target you have to copy the generated file to the proper place for udev-rules in order to take effect.

integrity_check   Performs an integrity check on the SpartanMC installation. This will run an automated sequence covering most of the functionality of the SpartanMC-SoC-Kit. The sequence will start with calling **make unconfigure** to get a clean installation directory. The next actions cover all setup steps followed by the creation of a test project. Finally, this project will be synthesized. If *PLATFORM* specifies any other value than 'nohw', the design will be implemented on the corresponding target platform. Otherwise, the sequence will be complete after bitfile generation. To get a list of supported platforms, call **make integrity_check** without any option. If the described sequence completes, the SoC-Kit most likely is properly installed and configured. If not, there may be a configuration problem or a functional issue concerning the toolchain. The test sequence will abort with an error message in that case.

man   Displays the SpartanMC manpage denoted by *MANPAGE*. Ommitting *MANPAGE* is equal to **make man spartanmc** and will show this manpage.

reconfigure   Runs **configure** with the same options and relevant environment variables as the last time the configure was explicitely invoked via the command line.

setup   Builds or updates all required components of the SpartanMC-SoC-Kit from the corresponding sources. Components that any other make targets depend on are automatically built when invoking that target (e.g. manpages are generated from the users manual sources when invoking **make man**).

unconfigure   Removes all files generated by **make** and**configure**. After running **make unconfigure** , your SpartanMC installation will be left in the same state as just after a fresh install.

what   Shows a list containing all currently available targets and a short description. The availability of some targets may depend on your host system configuration or the current state of your SpartanMC installation or current project.

SEE ALSO

MANPAGE – SPARTANMC-HEADERS(7)

NAME

spartanmc-headers – SpartanMC header files for firmware development

DESCRIPTION

The various functions implemented in the SpartanMC C library (see *spartanmc-libs*) are defined in a number of header files located at **spartanmc/include/**. This path is part fo the standard include path of the SpartanMC-GCC. The following sections describe use and organization of the header files.

LIBRARY HEADER FILES

The following header files define general support functions and macros as well as support functions for access peripheral components:

bitmagic.h	Macros for bit manipulation such as set/clear/toggle bit at a certain memory location.
ddr.h	Support functions for the MCB (Memory Controller Block) in Xilinx devices (mcb_peri_interface). Requires *libperi*.
interrupt.h	Support functions for interrupt controller peripheral (intctrl, intctrl_p). Requires *libinterrupt* or *libinterrupt_p*.
led7.h	Support functions for a 7-Segment Display connected to the SFR_LEDS special function register of the processor core. Requires *libperi*.
mul_high.h	Support function to access the high order word (bits 19-36) of a multiplication.
sleep.h	Function to delay execution by a certain number of clock cycles.
stdint.h	Integer type definitions.
stdio.h	Standard input/output functions such as printf.
stdio_uart.h	Support functions to use UART with stdio functions.
stepper.h	Support functions for stepper motor peripheral (microstepper). Requires *libperi*.
string.h	String and memory operations.
uart.h	Support functions for UART peripherals (uart, uart_light). Requires *libperi*.
usb.h	Support functions for USB11 peripheral (usb11). Requires *libperi*.

GENERATED PROJECT HEADER FILES

To allow easy access to the hardware components of any generated system there are a number of header files generated by the system builder *jConfig*. For details, see **peripherals.h** and **hardware.h**.

FILES AND DIRECTORIES

**spartanmc/
include/**   Header files for peripheral support functions

**spartanmc/
include/
peripherals/**   Header files defining structures and bit constants for peripheral register access

SEE ALSO

peripherals.h(3), **hardware.h**(3), *spartnmc-libs*(7)

AUTHORS

Copyright (c) 2011, 2012 Dresden University of Technology, Institute for Computer Engineering, Chair for Embedded Systems.

Written by Markus Vogt

MANPAGE – HARDWARE.H(3)

NAME

hardware.h – Header file populating hardware implementation parameters for low level hardware access

SYNOPSIS

#include <system/hardware.h>

DESCRIPTION

The project specific generated header file **hardware.h** populates a number of key-value pairs reflecting all synthesis parameters passed to the hardware implementation process. This allows the firmware to be aware of certain features and parameters concerning the hardware platform it runs on.

The actual keys available depend on the system configuration specified in the system builder *jConfig*. Basically, each value choosen at the tab *Parameters* of each hardware component is mapped to a **#define** using the form

#define <KEY> <VALUE>.

Refer to the hardware documentation for details about the actual parameters defined by a certain hardware component.

KEYS

The list below gives an overview about the general format of available keys populated by **hardware.h**.

SB_<instance_name>_<parameter_name>

> Value of hardware parameter <parameter_name> of module instance <instance_name>. E.g., the base address (parameter *BASE_ADR*) of module *uart_0* would be **SB_UART_0_BASE_ADR**.

SBI_REGION_<instance_name>_<suffix>

> Provides information about the address space occupied by a certain hardware module. This only applies to processor instances and peripheral modules implementing DMA. **<suffix>** can be one of **MIN_ADDR**, **MAX_ADDR** or **BYTES** respectively giving the lower

or upper address boundaries or the number of bytes occupied by the components memory.

SBI_VERSION   System builder version, which is currently 2.

SBI_CORE_ID   18-bit hexadecimal checksum of the current hardware design. Used to match a given firmware binary against a certain hardware design when using the bootloader (see *spmc-loader*(1)).

i_bits   The number of interrupt lines provided by the interrupt controller (*intctrl* or *intctrl_p*), if any. When no interrupt controller is present, the value for **i_bits** is 0.

VALUES

The form a value is represented depends on the parameters value type as defined by the system builder *jConfig* or read from the respective module description. All values are in fact mapped to integer constants. To deal with float and string values, symbolic constants are used.

Integer values

>Decimal and hexadecimal integer values are represented straight forward as shown in *jConfig* (e.g. **23**, **0x42**). Binary numbers are represented using their respective hexadecimal notation.

Float values

>Float value parameters defined by the system builder are represented by symbolic constants of the form **SBFLOAT_<int>_<frac>**. E.g, the float number **2.68** woule become **SBFLOAT_2_68**. Note that this technique only allows for test of equality regarding a certain parameter. Performing real float arithmetics is not possible, which rather is limited by the fact that the SpartanMC currently does not support floating point in any way.

Boolean values

>Boolean values are represented by integers of value **0** or **1** respectively standing for **false** or **true**. Constants of the form **SBBOOL_...** map the symbolic representation for each boolean parameter to their respective numeric values **0** or **1**. E.g., a boolean parameter with the symbolic meaning of **YES** or **NO** will provide the constants **#define SBBOOL_YES 1** and **#define SBBOOL_NO 0**. This allows you to use symbolic constants similar to as shown in the system builder when testing for values of boolean parameters.

String values

>String values are mapped to symbolic constants of the form **SBSTRING_<value>**, where **<value>** is replaced by the original string value in upper case. All characters not allowed in a constant name are replaced by a underscore (_). E.g., the value of parameter VENDOR_STRING =

"TU Dresden" of component *usb11_0* would become**#define SB_USB11_VENDOR_STRING SBSTRING_TU_DRESDEN**. The symbolic constant **SBSTRING_TU_DRESDEN** is mapped to an arbitrary unique hexadecimal value.

FILES

**<project_dir>/
system/
hardware.h**   Header file to include for access to hardware parameters

**<project_dir>/
system/
<subsystem_name>/
hardware.h**   Actual header file defining hardware parameters for the respective system. Included by **hardware.h** depending on the actual subsystem the firmware is built for.

SEE ALSO

peripherals.h(3)

AUTHORS

Copyright (c) 2011, 2012 Dresden University of Technology, Institute for Computer Engineering, Chair for Embedded Systems.

Written by Markus Vogt

MANPAGE – PERIPHERALS.H(3)

NAME

peripherals.h – Project header file for access to peripheral components

SYNOPSIS

#include <system/peripherals.h>

DESCRIPTION

For any SpartanMC project, the system builder *jConfig* generates a header file providing the interface to all peripheral components present in your system. Variables pointing to the respective I/O and/or DMA memory base addresses will be automatically provided for each peripheral instance.

For a particular peripheral instance the name of the variable will be the respective identifier as shown in the system builder *jConfig* converted to UPPER CASE (e.g. *UART_LIGHT_0*). Each such variable will be a typed pointer tailored to the register I/O space of the particular peripheral. In case a component offers DMA space you will get another variable named <PERIPHERAL_NAME>_DMA pointing to the peripherals DMA base address.

The header files defining the respective variable types can be found at **spartanmc/include/peripherals/** (see section below for details). All files required for your systems peripherals will be automatically included via **peripherals.h**.

IMPLEMENT CUSTOM PERIPHERALS

For each type of peripheral component a header file is required at **spartanmc/include/peripherals/** declaring the particular data types (e.g. a struct) for register I/O and DMA access. The header files name must be the same as the peripherals hardware type.

If you add a custom peripheral component to the SpartanMC-SoC-Kit make sure you provide the corresponding header file. For a component named e.g. *my_peri* a file named **my_peri.h** is required. Within this file the following type declarations are expected to be found:

typedef ... my_peri_regs_t; /* register space access */

typedef ... my_peri_dma_t; /* DMA space access */

Note that in case your peripheral does not implement registers or DMA space the respective type declaration may be ommitted. Basically, the interface to a peripheral component may be a pointer to an unsigned integer. In that case, the type definitions may look like the following:

typedef unsigned int my_peri_regs_t;/* register space access */

typedef unsigned int my_peri_dma_t;/* DMA space access */

Note that there is no explicit pointer- or array-like declaration. The point where the pointer comes in is at the variable instatiation in the generated header file.

To interface more complex peripherals it is wise declaring a structure with descriptive names for the particular registers. Additionally to the type declaration the header file may define bit constants to simplify bit wise access to the registers.

High level support functions operating on the peripherals registers and DMA space should be defined in arbitrary named header files located at**spartanmc/include**. The resepective implementation of such functions should be part of*libperi*, but could virtually be implemented in any other library.

FILES

<project_dir>/
system/   Header file to include for access to generated peripheral variables
peripherals.h

<project_dir>/
system/   Actual header file defining peripheral variables for the respective
<subsystem_name>/system. Included by **peripherals.h** depending on the actual
peripherals.h   subsystem the firmware is built for.

SEE ALSO

hardware.h(3),*spartanmc-libs*(7)

AUTHORS

Copyright (c) 2011, 2012 Dresden University of Technology, Institute for Computer Engineering, Chair for Embedded Systems.

Written by Markus Vogt

MANPAGE – SPARTANMC-LIBS(7)

NAME

spartanmc-libs – SpartanMC software libraries and library build system

DESCRIPTION

The SpartanMC processor core comes with a number of supporting C libraries. The library functions are available for the firmware on the target system. Some library code is essential such as startup code or interrupt handling routines. The remainder of the library offers functions to the user to simplify access to peripheral components.

To use functions from a certain library, the corresponding header file must be included (see *spartanmc-headers*) in your source code and the linker must be told to include the library into the firmware binary. The latter is achieved by adding the library name to the list of link libraries in *config-build.mk* in the firmware directory. Note that pointer variables for access to peripheral registers are available by including *peripherals.h* (see **peripherals.h**).

To optimize the size of the resulting firmware binary, each library function is implemented in a separate source file. This allows the linker to remove all functions that are never called.

LIBRARIES

The following libraries are currently available for the SpartanMC processor core:

libc   System calls like printf, sleep, etc. Automatically linked with each project.

libgcc   Support function for the compiler. Automatically linked with each project.

libinterrupt   Required interrupt handler and support functions for default interrupt controller (intctrl). Could not be used together with **libinterrupt_p**.

libinterrupt_p   Required interrupt handler and interrupt support functions for interrupt controller with priority (intctrl_p). Could not be used together with **libinterrupt**.

libperi   Support functions for various peripheral components such as USB, UART and others (also see **peripherals.h**).

librtos   Real Time Operating System. Can not be used together with **libstartup**.

librtos_interrupt Interrupt support for**librtos**. Can not be used together with**libstartup**.

IMPLEMENTING NEW FUNCTIONS

Extending an existing library

To add a function to an existing library, create a new file in the corresponding source folder in**spartanmc/lib_obj/src/<library_name>/**. To get a smaller binary through link time optimization, make sure to implement each function in a separate source file. The type of source file can be either C (*.c) or Assembler (*.s).

To make your new function known to the compiler, create or edit a header file in**spartanmc/include/**. Your implemented function can use other library functions. See section *Library build system* below on how to specify dependences between libraries.

Creating a new library

An entirely new library is create in a seperate folder named**spartanmc/lib_obj/src/ <library_name>/**. As described above, create source files in that new folder to implement your library functions.

The new library must be included into the build system. Edit the file *spartanmc/lib_obj/ Makefile* for that purpose. See below for details on the library build system.

The library build system

All source files for one certain SpartanMC library are placed in a dedicated directory. The location of the source files is**spartanmc/lib_obj/src/<library_name>**. Possible source file types are C (*.c) and assembler (*.s).

The library build process is controlled by the makefile**spartanmc/lib_obj/Makefile**, where the following variables are of interest:

LIBS List of libraries to build. Each name specified must correspond to a source directory**spartanmc/lib_obj/src/<library_name>**. All library names are specified without the prefix*lib*.

OBJ_DIRS List of directories with additional object files. These files are compiled but not explicitly bundled into a library archive file (*.a). Other library functions can use the resulting objects as dependences. This is useful for helper functions which could not explicitely associated to a certain library.

DEPS_<library_name>

Specifies dependences for each library, if required. Each object file specified here is included in the library archive (*.a) additionally to the original library code. Valid objects are either from another library or from a directory specified via variable **OBJ_DIRS** (see above).

FILES AND DIRECTORIES

**spartanmc/
lib_obj/src/
<library_name>**   Source code for all functions of library <library_name>

**spartanmc/
lib_obj/
Makefile**   Makefile controlling the library build process

SEE ALSO

spartanmc-headers(3),*spmc-loader*(1),*startup_loader*(3)

AUTHORS

Copyright (c) 2011, 2012 Dresden University of Technology, Institute for Computer Engineering, Chair for Embedded Systems.

Written by Markus Vogt

MANPAGE – SPARTANMC-STDIO(7)

NAME

spartanmc-stdio – SpartanMC standard Input / Output

DESCRIPTION

Standard input and output is handled via FILEs. They reference a peripheral or section of memory and functions to perform input/output. They may be designated as default input / output, so that they are used whenever no explicit FILE is specified.

For each Peripheral that supports access via FILE, jConfig automatically generates a declaration of a matching FILE structure. It has the same name as the peripheral, with "_FILE" suffixed.

USING FILES

Output functions like *printf* need an output FILE to be set. Your code needs to define *stdout* similar to this:

```
FILE * stdout = &UART_LIGHT_FILE;
```
Input functions like *getchar* need an input FILE to be set. Your code needs to define *stdin* similar to this:

```
FILE * stdin = &UART_LIGHT_FILE;
```

IMPLEMENTING CUSTOM FILES

FILE is a *struct* containing a base pointer and a number of function pointers. Each function pointer references a function that performs some input or output action. When called, it gets passed the base pointer and additional function-specific arguments if applicable.

base_addr   Base address of peripheral / memory location used to input/output

send_byte   Output a single byte. If the transfer cannot be initiated directly (because of full hardware buffers) the function blocks.

receive_byte   Input a single byte. If there is no input byte available (because of empty hardware buffers) the function blocks.

receive_byte_nb   Input a single byte if there is one available. If there is no buffered input byte, signal that via return code. The received byte is output via its pointer argument.

MANPAGE – PRINTF(3)

NAME

printf – formatted string output

SYNOPSIS

#include <stdio.h>

int printf(const char **s***, void ****arg1***, void ****arg2, ...***);**

DESCRIPTION

The function**printf**() produces formatted string output according to the format specifications found in the standard C documentation with respect to the limitations described below. It expects its argument *s* to be a null-terminated character string followed by any number of arguments serving as input values for the output format conversion. The argument *s* must not be NULL. Output is sent to the FILE set in stdout.

Return Value

This function returns the number of characters that it printed.

Conversion specifiers

The following standard conversion specifiers are supported (see standard C printf documentation for details):

d,**u**	Decimal number in signed (**s**) or unsigned (**u**) notation
x,**X**	Hexadecimal number using lower or upper case notation
o	Octal number
s	String
%	Percent ('%') character

The following additional non-standard conversion specifiers are supported:

b	Interprets the given argument as *unsigned int* and produces an output string in binary notation.

Flags, field width, precision, length modifiers

The only supported *flag* is **0** for leading zeroes. *Field with* is supported with a maximum value of **18**. Specifying greater values may lead to undefined behaviour. *Precision* and *length modifiers* are not supported.

AUTHORS

Copyright (c) 2011, 2012 Dresden University of Technology, Institute for Computer Engineering, Chair for Embedded Systems.

Written by Markus Vogt

MANPAGE – DEBUGGING(3)

Library

The library contains the debugging stack, that talks to GDB. For SpartanMCs without Hardware Debugging Support use "debugging_soft" (Only Memory Breakpoints supported), when Hardware Support is present, use "debugging" to make use of the additional features and remove support for memory breakpoints.

Changes to firmware/config-build.mk:

Add "debugging" or "debugging_soft" to the LIB_OBJ_FILES property.

Add "debugging_traptable" to the LIB_AS_FILES property or provide your own version of that trap table.

Add "-g3" to the CFLAGS and LDFLAGS properties. It is advised to use "-O0" as optimization level.

Changes to codefile containing main():

#include <debugging.h>

Add FILE * debugIO = &<PERI>_FILE; as a global variable with <PERI> being the name of the peripheral to use for debugging IO.

Hardware Support

Hardware support enables stepping, hardware breakpoints and watchpoints. To enable hardware support activate the boolean flag `HARDWARE_DEBUGGING` for the desired SpartanMC core in JConfig.

The default trap mapping should be fine, number of break-/watchpoints can be configured.

Hooks

The Debugging Stack defines some hooks to ease integration with other peripherals. Simply define the functions as listed below.

void debuggerAfterStop();
　　　　　　Gets called when the program gets stopped.

void debuggerBeforeContinue();
　　　　　　Gets called when the program is about to be resumed.
When using the debugging stack together with memory guards, they should probably be disabled while the program is stopped. Otherwise, GDB cannot read the instruction memory. Code to do this may look like this:

```
int debuggerMemguard;
void debuggerAfterStop() {
    debuggerMemguard = memguard_is_enabled(&MEMGUARD_0);
    memguard_enable(&MEMGUARD_0, 0);
}
void debuggerBeforeContinue() {
    memguard_enable(&MEMGUARD_0, debuggerMemguard);
}
```

Building GDB

If following the current setup manual, GDB (*spartanmc-gdb*) is part of a standard setup.

GDB Usage

Use the following command to start GDB:

```
spartanmc-gdb --baud 115200 <path to elf file>
```

The option `--baud 115200` is only needed if the selected peripheral for debugging IO is UART.

Connect to the target device with `target remote /dev/ttyUSBx`

Alternatively, you can do both in a single command:

```
spartanmc-gdb --baud 115200 --ex="target remote /dev/ttyUSBx"
<path to elf file>
```

The SpartanMC-GDB is somehow special. Due to the special bitwidth of 18-bit for Data and Instructions, only a subset of commands are supported. Nevertheless, they should be enough to cover the most relevant use cases. The following table shows the supported commands:

Command	Explanation	Example
backtrace (bt)	Prints a backtrace.	bt
hbreak	Sets a hardware breakpoint.	hbreak main.c:30 - set a breakpoint at line 30 in main.c
delete	Deletes a breakpoint	delete 1 - deletes first breakpoint
continue (c)	Continues execution until the next breakpoint is hit.	c
print (p)	prints a variable or the result of an expression	• p input[0] - prints first element of input • p/c message_vec[0] - prints first element of message_vec as character • p reg - prints variable reg

Command	Explanation	Example
		• `p &reg` - prints address of reg
`watch`	Inserts a write watchpoint for a given address or variable. Execution will be stopped, when the value of the variable changes.	`watch reg` - watches variable reg.
`step ( s )`	Step one instruction. Steps *into* function calls	`s`
`next ( n )`	Step one instruction. Steps *over* functions calls	`n`
`ptype`	Prints type of variable	`ptype reg`

Table -158: GDB commands supported by SpartanMC.
Shorter versions of a command are printed in parentheses.

Note: Reading strings (char arrays) as a whole is not supported and will return non-sense. Nevertheless, accessing single elements in a char array is possible.

Note: SpartanMC-GDB can also read peripheral register and structs.

Note: One advantage of the terminal based GDB over most GUI implementations is the use of *inferior calls.*

In order to insert Breakpoints, the GDB modifies the system's instruction memory. Furthermore, GDB host and client need to be kept in sync in order to work properly. Therefore, resetting the system requires a little more care. Before hitting the reset button, it is advised to delete all breakpoints, stop the GDB on the host computer and to restart it afterwards. Note that pressing the reset button doesn't create a clean instruction or data memory. If a corrupt instruction or data memory is suspected, the only option is to reprogram the FPGA board.

MANPAGE – SPMC-LOADER(1)

NAME

spmc-loader – update firmware on SpartanMC systems

SYNOPSIS

spmc-loader *PORT SYSTEM_ID SPH_FILE*

DESCRIPTION

Overview

Modifies program memory content on the current SpartanMC system by uploading an updated version of the firmware image using a serial port connection (UART).

In normal use cases *spmc-loader* will be invoked by *make upload*. All options described below will be set to proper values corresponding to the current project in that case.

Upload process

The upload process uses a serial UART connection to transfer new program memory content to the SpartanMC processor core of the target system. To enable the ability to receive bytes and store them into memory, a special startup routine need to be compiled into your initial firmware image. This is accomplished by specifying *startup_loader* instead of *startup* in the list of library objects. See *startup_loader* for details on the upload process.

OPTIONS

PORT Specifies the serial port the UART of the target system is connected to.

SYSTEM_ID 5-digit hexadecimal number of the form 0x12345 identifying the hardware design currently present on the FPGA. This number is sent to the target device and must match the number stored within your current hardware design during synthesis. The actual value is determined by calculating a checksum over all hardware synthesis parameter values present in the design. If both numbers do not

match, the firmware upload is aborted. This mechanism prevents uploading a firmware image to the wrong hardware platform.

SPH_FILE   Specifies the *.sph file to upload to the processors program memory.

SEE ALSO

startup_loader spartanmc-project sph

MANPAGE – STARTUP_LOADER(3)

NAME

startup_loader – startup system library with support for updating of program memory content

SYNOPSIS

Link with *-lstartup_loader*

Can not be used together with *-lstartup*

DESCRIPTION

Overview

Provides startup code that allows for replacement of the processors program memory without re-synthesizing the hardware design.

The tool *spmc-loader* implements the mechanism described below on host side to support firmware upload.

Data format

The SPH file format is used to transfer the binary image to the target. For more information, see *sph*.

Upload process

The upload process is started after system reset when requested by the host. When linked with *-lstartup_loader* the startup routine will check for such request and enter the upload routine. Otherwise, the program currently present in memory will be executed as usual.

The initial request is followed by a handshake mechanism to avoid accidental corruption of memory when the hosts UART is transmitting some other unrelated data during system reset. If the handshake fails at any stage, the loader routine exits. Normal program execution follows in that case.

After completing the upload process the target system requires another reset to start execution of the uploaded program.

SYMBOLS

LOADER_UART_BASE
> Base address of UART peripheral used for data transfer. The only supported UART hardware currently is *uart_light* .

LIMITATIONS

The upload process can update all memory regions including DMA areas with the following limitations:

The loader code itself cannot be updated. The startup code placed before the loader code (at lower adresses) must not change in size. Both cases will be detected ba the upload routine and reported to the user.

SEE ALSO

spartanmc-project sph

40. microStreams

microStreams is a Cetus based tool, that allows the transformation of the source code form a single threaded application to multi-core streaming pipeline program. It is evident that constructing processing pipelines is only useful for repetitive tasks whose throughput shall be increased. Thus, this approach is very well applicable for microprocessors running repetitive tasks on bare metal. So instead of trying to run several instances of this application and count on data parallelism, microStreams extracts different steps as dependent tasks of the application, constructing a data pipeline. However, an additional parallelization in form of a superscalar pipeline is additionally possible.
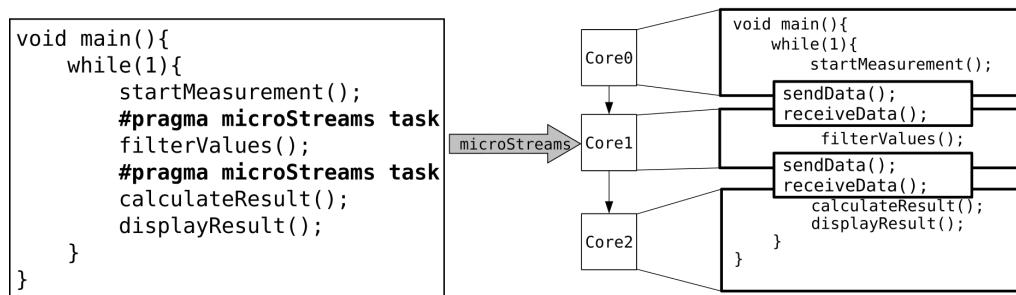

Figure 40-57: Parallelizing Source-Code with microStreams

40.1. Usable Pragmas

In order to successfully transform the application, the user has place few pragma annotations to indicate different tasks in the application. The possible annotations for microStreams are in the form of `#pragma microstreams task [optionals]`. While the [optionals] tag can hold one or more (space separated) of the follwing values:

- `in(variable_name, other_variable)` : Sets the variables listed in the brackets as forced input variables for the task. It is in practice rarely necessary to use this argument, since the variables are in most cases automatically recognized.

- `out(variable_name, other_variable)` : Sets the variables as forced output variables of the task. For details see the previous in optional

- `replicate(nr_of_replications)` : Replicate this task as many times as specified. The tasks will then work in parallel and the replicated tasks can handle new workloads while the original task is still busy processing.

- `end` Ends a task pragma explicitly. Otherwise tasks are ended by scope ends or new task pragmas. This can not be used in combination with other [optionals]

As shown in the example in the following Listing. Pragmas can be placed before function calls, loops or ordinary statements. The created tasks code will reach until the next microStreams task pragma, the end of the scope where the pragma was placed in or until an explicit task end pragma is reached. However the task end pragma is rarely necessary.

```c
#pragma microstreams task
void foo(int x){
    int ret = 4;
    #pragma microstreams task
    ret++;
    return 4;
}

void main() {
    int c[10], i;
    #pragma microstreams task
    for(i = 0; i < 10; i++) {
        c[i]=i*i;
    }
    sum(&c);
    #pragma microstreams task
    print(&c);
    #pragma microstreams task end
}
```

40.2. Processing Pipeline

A processing pipeline is automatically created from the pragmas if the data dependencies allow it. microStreams will split the source code at the declared pragmas and create a dependency graph. The dependency reflects the usage of common variables in the tasks. Based on the dependency graph a communication infrastructure between tasks is created to exchange the non exclusive variables. Each task will be mapped to one processing core and a hardware communication infrastructure with simple FIFO buffers is designed. An example for such a system can be seen in the following Figure. It was demonstrated that this methodology is applicable despite of exchanging big data arrays between the tasks in http://ieeexplore.ieee.org/document/7518530/ .
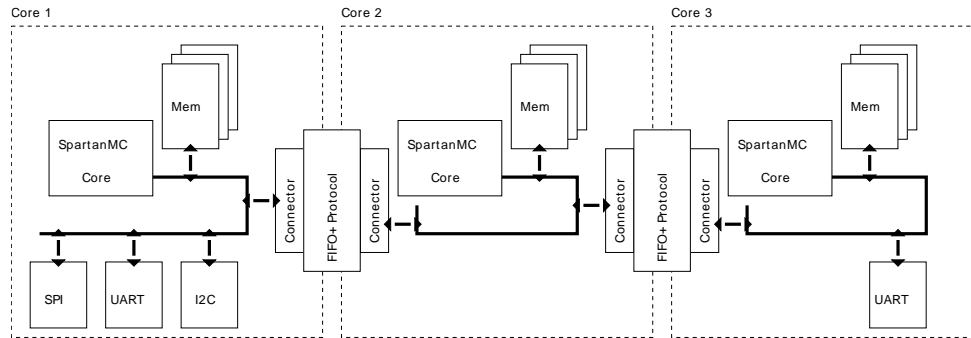
**Figure 40-58: A sample SpartanMC based multi-core system
consisting of three cores and several peripheral components**

40.3. Performace Evaluation

In addition to the automated firmware splitting the tool also offers evaluation techniques to help users judge the created design. The tool can create an environment to measure the time spend for task execution and data transmission to the next task with cycle counters. With those measurements the user can determine if the pipeline stages are balanced and see the communication overhead. The performace evaluation uses performance coutners for each core, and the results of each core will be send to the first core which contains the so called base task via a global memory.

40.4. Created Files

As output, microStreams delivers several firmware files and a HW system description. The system description (hardware.xml) can be read into the system builder (jConfig) via the import flag and additionally selecting a target flag specifing for which FPGA Board to build. Note, that the import is able to add all necessary components and mostly connect them properly. However the configuration of the components such as for example RAM Blocks for memory is set to a default value and should be revised manually (based on the single core design). A good starting point is to set the RAM amount of each core to the one of the single core design and then start reducing and checking if the system still works.
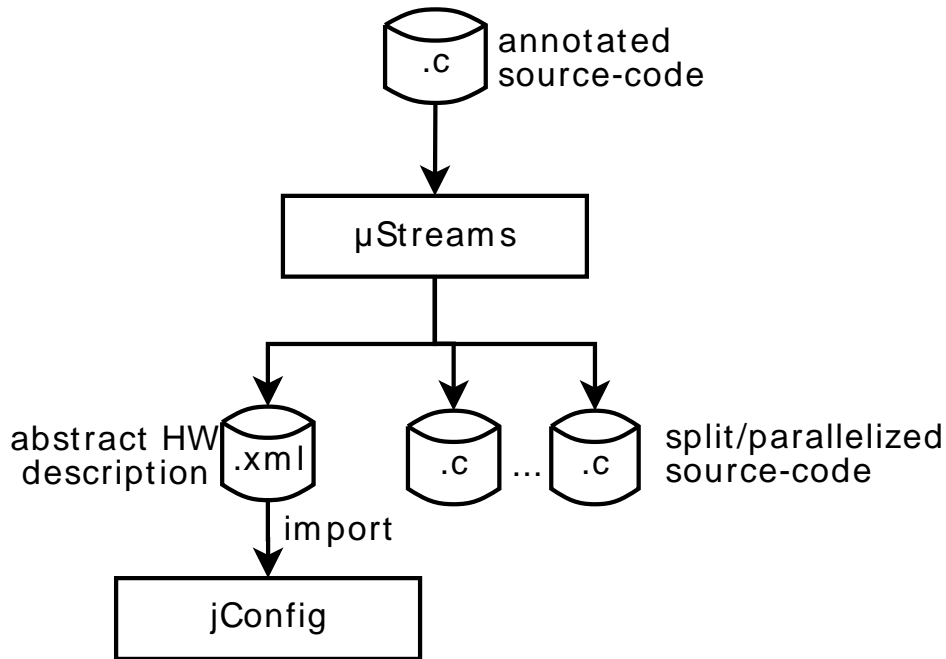
Figure 40-59: microStreams toolflow

40.5. Commandline Options

microStreams can be called in a project directory with: make microstreams +args="[options]" +firmware="firmware folder to analyze" . The following options are possible:

- `-a, --autolibspartanmc` : Automatically detect SpartanMC lib folder. Default: false.

- `-e, --evaluate` Add constructs for a performance measurement evaluation on each core. Default: false

- `-h, --help` Show this help message Default: false

- `-i, --inspect` Shows the generated model with the tasks and task dependencies in a graphical window. Default: false

- `-l, --libs` Give path to additional libraries i.e. header files. Paths are separated by comma. Default: []

- `-g, --no-global-mem` Tells weather we want to move variables to global memory or not. Creates HW and SW. Default: false

- `-p, --project` Should point to the directory where the spartanmc project to be modified resides. Default: .

- `-s, --sim` Generate hardware for a simulated target. Default: false

41. microStreams - AutoPerf & SerialReader

AutoPerf is a simple but quite useful tool, based on parts of microStreams, for profiling applications. AutoPerf expects the source code of an application to profile as input and injects calls to the cycle counter of the SpartanMC processor into it. Those calls will be injected before and after function calls, loops and successive code blocks, as shown in the following listing. By default only the body of the main function will be profiled. With a pragma (`#pragma autoperf`) ahead of another function or inside another functions body the user can profile different parts of the firmware as needed. After running the instrumented code on the device, the cycle counter results can be dumped via UART. The program AutoPerf-SerialReader can read the UART output, separate system out and performance results and write the performance-counter report into a CSV file. This report contains an exact application profile showing which part of the source code took how many cycles to execute. An example of such an output can be seen in the following table. The report contains a field specifying the location of the measurement, the source code line where the measurement was started in the original source (not in the modified program) and the execution time for this step. The application profile is an important step to choose a good pragma placement with microStreams.
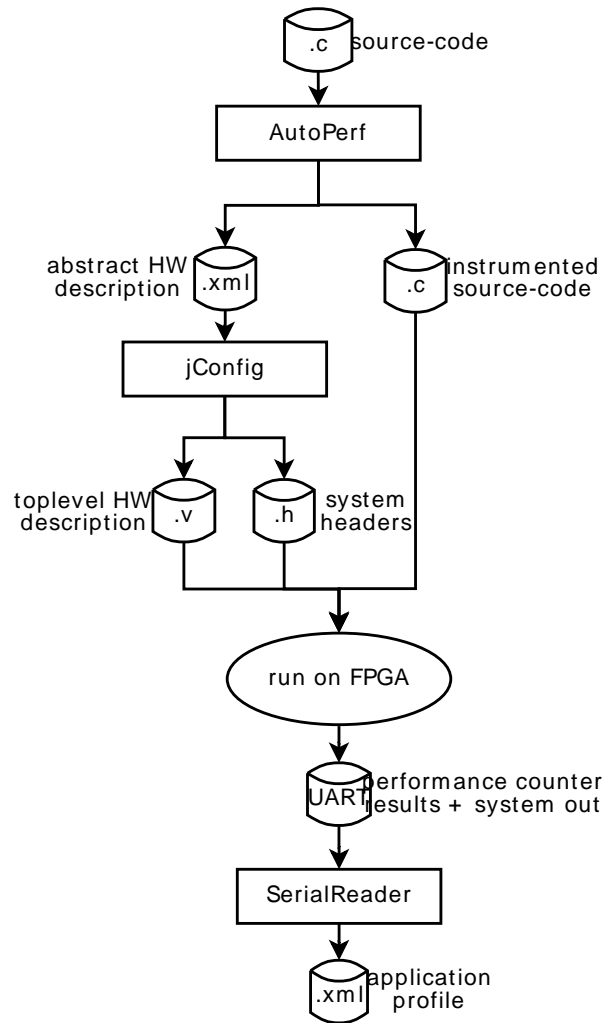
Figure 41-60: AutoPerf workflow

Input Source-Code:

```
void main() {
    int c[10], i;
    for(i = 0; i < 10; i++) {
        c[i]=i*i;
    }
    sum(&c);
    print(&c);
```

Instrumented Source-Code:

```
void main() {
    perf_auto_init();
    perf_auto_start();
    int c[10], i;
    perf_auto_stop(0, perf_results);
    perf_auto_start();
    for(i = 0; i < 10; i++) {
```

```
      c[i]=i*i;
   }
   perf_auto_stop(1, perf_results);
   perf_auto_start();
   sum(&c);
   perf_auto_stop(2, perf_results);
   perf_auto_start();
   print(&c);
   perf_auto_print(3, perf_results);
```

Source File	Function	Code line	Cycles
main.c	main	2	2
main.c	main	3	215
main.c	main	6	385
main.c	main	7	196

Table 41-159: Performance Report

41.1. Commandline Usage

AutoPerf can be called in a project directory with: make autoperf +args="[options]" +firmware="firmware folder to analyze" . The following options are possible:

- `-h, --help` Show this help message Default: false

- `--loopOpt` Switches Mode to profiling pragma ustreams loopOpt autoperf statements Default: false

- `--profile-arrays` Switches the report of array sizes on Default: false

- `-p, --project` Should point to the directory where the spartanmc project to be modified resides. Default: .

AutoPerf-SerialReader can be called in a project directory with: make serialreader +args="[options]" . The following options are possible:

- `-s, --array-size-dump` Specify the output File for the dumped array sizes. Default: arrays.csv

- `-a, --autoperf` Same as '-t autoperf'. Overwrites -t Default: false

- `-h, --help` Show this help message Default: false

- `-l, --loopOpt` Same as '-t loopopt'. Overwrites -t Default: false

- `-m, --microstreams` Same as '-t microstreams'. Overwrites -t Default: false

- `-o, --output` Specify the output File for the results. Default: results.csv

- `-p, --port` Set the serial port to listen to Default: /dev/ttyUSB0

- `-t, --target` Set the target/ Or from which application the performance results are generated Default: autoperf Possible Values: [microstreams, autoperf, loopopt]

42. VideoI420

The VideoI420 is a module, which uses a HDMI output to deliver a video signal. It needs some well defined clocks for the TMDS-Signal used by the HDMI wires and is configured through the peri-bus. A second connection to a memory is used to access the pixel data for displaying. The pixel data must be in the format I420(YUV), as this module was created for the VP8 decoder.
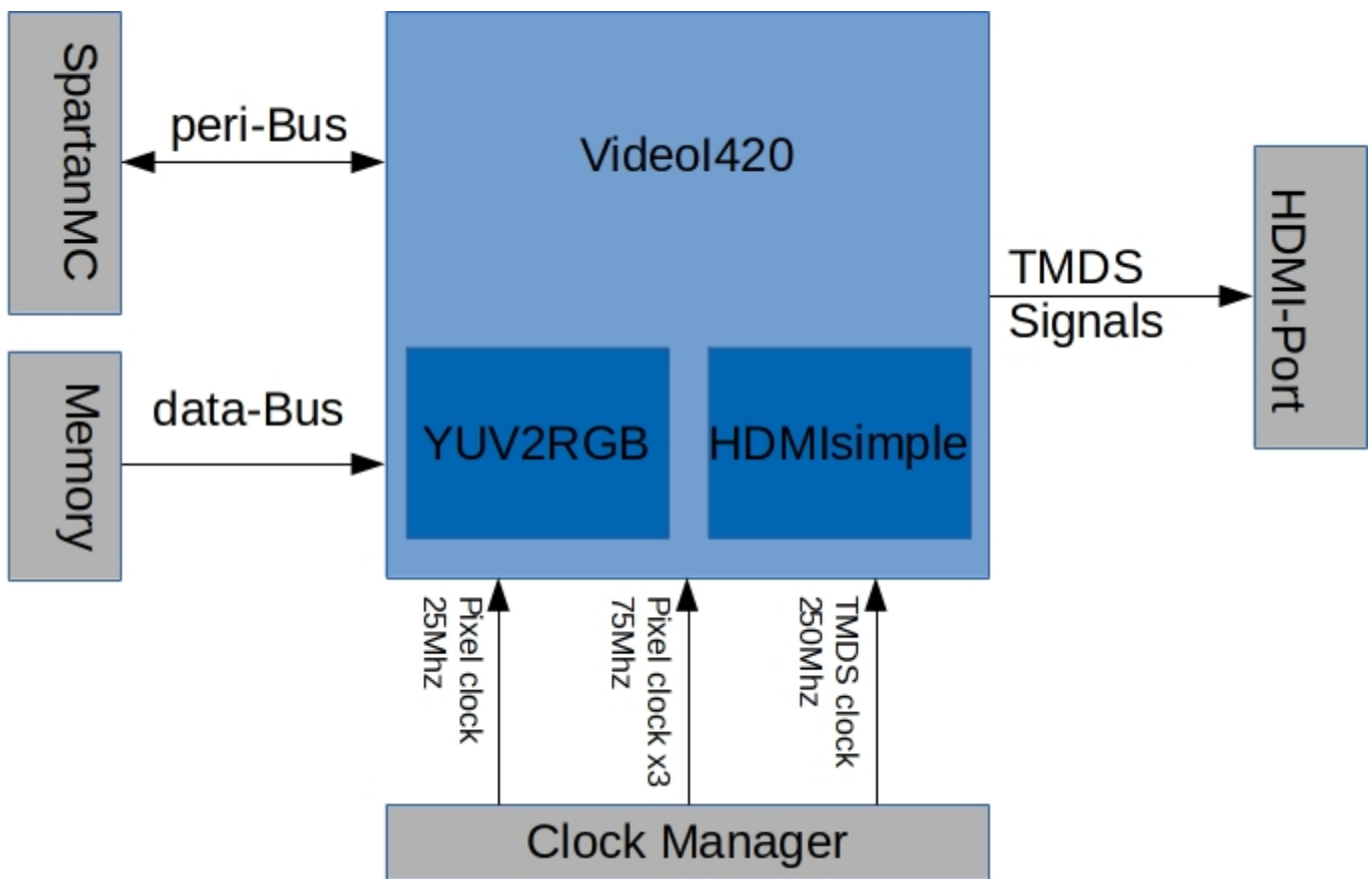
Figure 42-61: blackbox VideoI420

Parameter	Value	Information
PIXEL_CLOCK	24-25MHz	Should be fixed 25MHz as defined by the DVI specification. But allowing 24MHz for systems, where you may have an USB with 48MHz and want to have the clock as multiple of that clock(2*24MHz).

Parameter	Value	Information
PIXEL_CLOCK3X	3 times the PIXEL_CLOCK	This clock is used to convert the read YUV values from the memory to RGB values.
TMDS_CLOCK	240-250MHz	value is variable for same reason as for the PIXEL_CLOCK

Table 42-160: Parameters

Buses	Value	Information
peri-bus	/spartanmc_0@peri_bus	Connection to the SpartanMC for configuration settings.
mem-data-bus	...any memory-bus module	-
TMDS_CLOCK	240-250MHz	Connection to any memory, which contains the images for display. The memory must be able to deliver Pixel Data with the PIXEL_CLOCK3X clock.

Table 42-161: Bus Connections

Clocks	Value	Information
pix_clk	24-25MHz	Is used by the module to realize the TMDS signal
pix_clk3x	3 times the PIXEL_CLOCK	Is used by the yuv2rgb converter and is the access speed for the pixel data
tmds_clk	240-250MHz	Is the frequency inside the wire for the TMDS singal

Table 42-162: Clock Connections

Module Pin	FPGA Pin
TMDSp_clk	PACKAGE_PIN T1 IOSTANDARD TMDS_33
TMDSn_clk	PACKAGE_PIN U1 IOSTANDARD TMDS_33
TMDSp0	PACKAGE_PIN W1 IOSTANDARD TMDS_33
TMDSp1	PACKAGE_PIN AA1 IOSTANDARD TMDS_33
TMDSp2	PACKAGE_PIN AB3 IOSTANDARD TMDS_33
TMDSn0	PACKAGE_PIN Y1 IOSTANDARD TMDS_33
TMDSn1	ACKAGE_PIN AB1 IOSTANDARD TMDS_33
TMDSn2	PACKAGE_PIN AB2 IOSTANDARD TMDS_33

Table 42-163: TMDS Singals - only tested for Nexys-Video

42.1. Software Interface

To use the VideoI420 you need first to initialize the module with the method:

- `void videoI420_init(&VIDEOI420_0, PIXEL_WIDTH, PIXEL_HEIGHT, YJUMP, pY1, pY2, UVJUMP, pU1, pU2, pV1, pV2, 3);`

You define the reference to the module, the size of the Video, the Y and UV Jumps of the memory layout, which is the difference of stride minus width, and the pointers to the ixel values. The scale is interpreted as no scale with value 0, double size with a value of 1 and tripple size with a value of 2. The given pointer address to the Y,U and V memory positions will be accessed through the memdata-bus.

After the initialization you can start and stop the video with the following methods:

- `void videoI420_start(VideoI420_regs_t* v, unsigned int buffer);`

- `void videoI420_stop(VideoI420_regs_t* v);`

Where the buffer parameter is 1 or 2 and chooses the first memory position (pY1,pU1,pV1) or the second memory position (pY2,pU2,pV2). You can always reinitialize the module with other parameters. The module will continue to display the given buffer again and again.

To make the module swap the memory position you use the method:

- `void videoI420_bufferReady(VideoI420_regs_t* v, unsigned int buffer);`

This will make the video module first finished the display of the current buffer, and then display the other buffer. After this method call it is possible, that both buffers are in use, as the current is displayed and the other will be displayed on the next frame. In such a situation you should not write in any of the buffers. You can ask the module which buffer is currently in use:

- `int videoI420_getFreeBuffer(VideoI420_regs_t* v);`

It will return 0, if both buffers are busy. This state will change automatically as soon as a new Frame begins to display, because the older buffer will be displayed and can be used again, while the new buffer is displayed. 1 or 2 as a return value means that buffer 1 or 2 is free and 3 means that both buffer are not in use, which is the case when the module does not display anything because it is not started or has been stopped.

For use-cases where the position of the image buffer changes frequently, as it is in the VP8 Decoder, due to the 4 fixed images, where everyone can be the next to be displayed, two further methods can be used, which are very simple:

- `void videoI420_next(VideoI420_regs_t* v, unsigned char* pY, unsigned char* pU, unsigned char* pV);`

- `void videoI420_next_block(VideoI420_regs_t* v, unsigned char* pY, unsigned char* pU, unsigned char* pV);`

You can define the memory pointer to the next Buffer direct in this method call. If you use this methods, you do not need to specify the pointers in the init method. But you do still need the init methods with all the others parameters. While the next method with the block will not return until the new image is displayed and therefore the old one is free to use, the other methods returns without waiting. This will save you some time but you will have to poll with the method getFreeBuffer(...) for not returning 0 if you want to reuse the old image buffer, as it may be still in use. This can become handy if you have to run some other tasks before reusing the old image and do not want to waste time.

43. Memdualported

The memdualported is a very simple module, which acts as a dualported memory. Basically it is the same as the SpartanMC local memory and behaves as a DMA. Both ports can have different clocks. Therefore the SpartanMC can write data in this memory, while another module can read this data with another clock. The direct use as a specified DMA module is not possible, because the aligment of the SpartanMC DMA ist not flexible enough. Due to the fact that this module uses pSelect for every BRAM the alignment is more flexible and you can instantiate it at every position in the address space with the minimal alignment of a BRAM size.
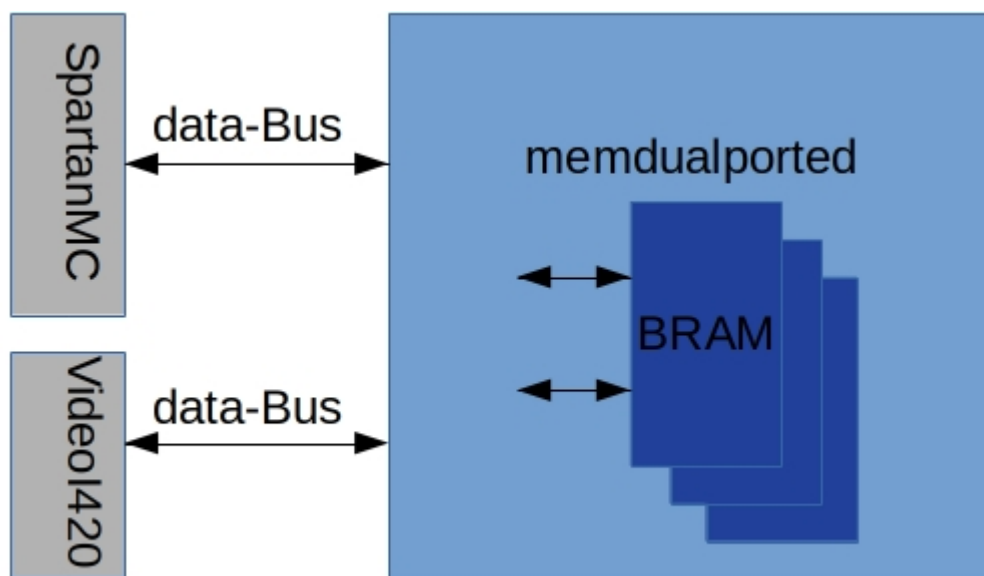

Figure 43-62: Blackbox memdualported

Parameters	Value	Information
RAMBLOCKS	e.g. 40	Number of Ramblocks you would like to use. 1 Ramblock is equal to (2*1024Bytes).
BASE_ADDR	e.g. 0xa000	This address should be set manually orientated on the autoScripted address space of the SpartanMC processor. It should begin with a position after the local memory

Table 43-164: Parameters

43.1. Use

This memory will be seen by the SpartanMC as further memory to the local memory and will not contain any firmware,software stack or interrupt table. Therefore this memory can be used completely for anything and is accessible by the SpartanMC and by any software running on it. The BASE_ADDR must be set manually, due to the fact, that the auto-script layout cannot handle it yet. This module is used in combination with the VideoI420 module and contains the images, which are read and displayed by the Video module and manipulated by the VP8 Decoder Software running on the SpartanMC.

44. LoopOptimizer

LoopOptimizer is a programm used to divide loops into multiple parts. These parts can be executed on multiple cores by running MicroStreams afterwards. AutoPerf can be used to improve the processing of loops.

44.1. Preparing your firmware

Before you run LoopOptimizer, you need to set LoopPragmas in your firmware. Loop-Pragmas need to be set directly in front of all loops that should be transformed. A Loop-Pragma looks like this: `#pragma microStreams loopOpt [number of splits] [optional parameters]` .

Possible parameters for `[number of splits]` (only one parameter at once):

- Number n: Direct specification by setting the number of splits.

- `maxCycles(` [Number as an Integer] `)` : Passes LoopOptimizer that none of the transformed loops should run longer than [Number as a Integer] Cycles. Needs AutoPerf Data to execute this.

Possible Parameters for `[optional parameters]` (multiple parameters possible at once):

- `autoperf` : Specifies that there is AutoPerf data available for this loop. Simultaniosly also serves as a Pragma for AutoPerf when running it in loopOpt mode.

- `distribution(` [n numbers seperated by comma] `)` : Passes LoopOptimizer to split the loops in a way that the runtime of the first transformed loop in relation to the runtime of all loops is approximatley equal to the relation between the first number of distribution() and the sum of all numbers. Can be executed with or without AutoPerf data (assumes equal runtime of all iterations and statements for the latter).

- `iterationVariable(` [name of a variable] `)` : Specifies which variable of the loop is the iterationVariable. Usually only necessary if LoopOptimizer terminated with a error referencing to this Pragma.

- `useSplitting` / `useFission` : Passes LoopOptimizer to use Splitting or Fission on this loop. Usually LoopOptimizer takes a good decision which shoule be used automatically. Splitting means that each transformed loops executes a part of all iterations, Fission means that each transformed loop executes a part of the body independent from the rest of the body.

44.2. Executing LoopOptimizer

Run `make loopOptimzer +firmware="<path>" +args="<args>"` to execute loopOptimizer. A new firmware will be created in the project folder with the transformed loops. The following parameters in `args` are possible:

- `-t, --taskpragma` : Passes LoopOptimizer to automatically set `#pragma microStreams task` between all transformed loops. See MicroStreams chapter for details.

- `--profile` [path]: Passes the Path to an AutoPerf profile that has been previously made by running AutoPerf in LoopOpt mode.

- `-p, --project` [path]: Specifies the path to the current project if not executed in this folder.

- `-a, --autolibspartanmc` : Specifies that LoopOptimizer should evaluate the `$SPARTANMC_ROOT` variable to find the SpartanMC Root location.

- `-l, --libs` [libaries]: Passes additional Libaries needed for executing the firmware, separated by comma.

- `-h, --help` : Show help message.
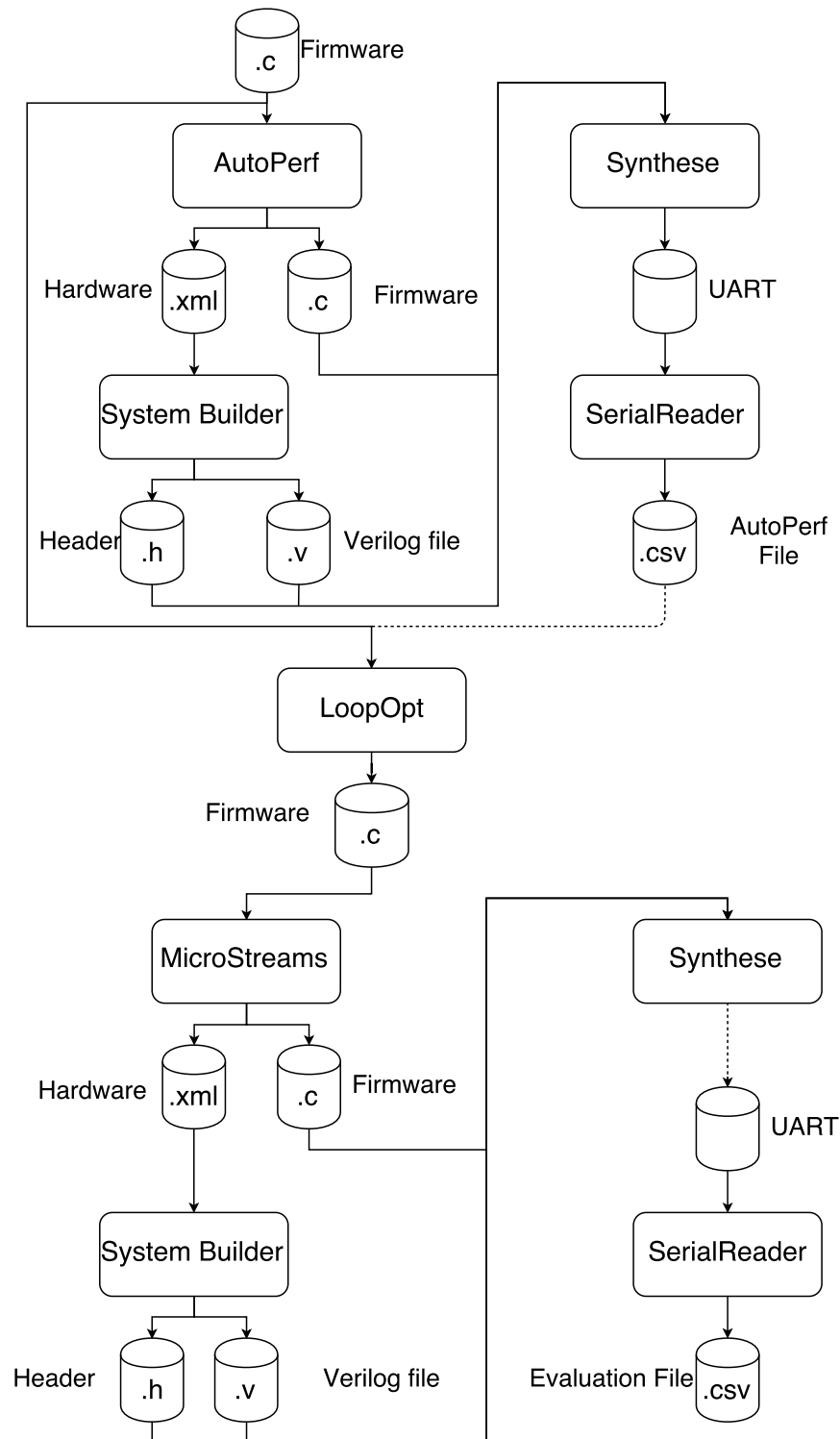
44.3. Example Workflow of LoopOpt

Figure 44-63: Beispielhafter Workflow eines einfachen Projekts

45. AutoStreams

AutoStreams is a java tool which finds a performance-optimal pipeline division for a sequential C program. It inserts task pragma annotations (`#pragma microstreams task`) to mark split points in the source code. The modified firmware can be used by microStreams in a next step to create the multi-core firmwares. For automatic pipelining, an AutoPerf performance profile is needed which contains the measured clock cycles of the individual statements in the code. Moreover, two result text files of an automated performance analysis of the core connector peripherals are required to estimate send and receive cycles for variables to be sent at a certain split point.

Pipelining can be performed in two different ways: either by specifying a maximum cycles count / time for each pipeline stage or by specifying a fixed count of stages for which the fastest possible pipeline is to be found.

45.1. Prerequisites

Apart from the input firmware / project, three files are needed for automatic pipelining:

- **performance profile:** AutoPerf adds start and stop statements to the input firmware. The modified firmware can then be programmed by the board. A string representation of the measured cycles is printed to the UART interface, which can be listened to by the AutoPerf SerialReader in order to write the corresponding CSV file (the performance profile). Usually, it has the name "results.csv".

- **send cycles text file / receive cycles text file:** The results of a performance analysis of the core connector transmission behaviour. Contains the measured send / receive cycles for ascending data sizes (in SpartanMC words) to be sent over the core connector, using the associated C functions. Currently contains results for data sizes between 1 and 133 words. The script for the automated measurement and for the creation of the text files is presently located under "tests/integrationtests/performance_analysis_core_connector". The file names of the results are "sendTimes.txt" and "receiveTimes.txt".

45.2. Usage and Command Line Options

AutoStreams uses the following command line options:

- `-p` , `--project` : Path to the SpartanMC project folder containing the firmware to modify. Default: current directory

- `--profile` : File path of the performance profile (CSV file), absolute or relative to SpartanMC project directory. Default: "results.csv" in the project or firmware directory.

- `--core-conn-perf-results` : Directory which contains the two result text files of the core connector performance analysis, absolute or relative to SpartanMC project directory. Default: Project directory.

- `-f` , `--clock-frequency` : Used SpartanMC clock frequency in MHz.

- `-m` , `--fpga-model` : FPGA model to determine the maximum frequency from.

- `-c` , `--cycles` : Maximum cycles for each pipeline stage.

- `-t` , `--max-pipeline-time` : Maximum execution time for each pipeline stage in ns. (Note: Cycles have priority if both -t and -c are specified.)

- `--stage-count` : Desired count of pipeline stages. (Can also be combined with -t / -c).

- `--connect-all` : Ensure that all stages of the resulting pipeline are connected by core connectors for synchronization. (Only insert pragmas at points where variables are transmitted from one core to another.)

- `-r` , `--report` : Write a detailed report (text file) of the created pipeline and the used configuration to the folder of the created firmware.

- `-ra` , `--report-all` : Like -r, but with information about (almost) all theoretically possible pipelines matching the specified requirement. Useful for comparing the resulting pipeline arrangement with other possible pipelines.

45.3. Example

Input firmware:

```
void main() {
    changeStreet(&addr);
    swapEntries(&bigArray);
    changeStreet(&addr);
    changeStreet(&addr);
    swapEntries(&bigArray);
}
```
Performance profile:

```
main.c:main:changeStreet(( & addr)):29;1113
main.c:main:swapEntries(( & bigArray)):30;1271
main.c:main:changeStreet(( & addr)):31;1113
main.c:main:changeStreet(( & addr)):32;1113
main.c:main:swapEntries(( & bigArray)):33;1271
```
Resulting AutoStreams firmware, using the argument `--stage-count 2` :

```
void main() {
    changeStreet(&addr);
    swapEntries(&bigArray);
    changeStreet(&addr);
#pragma microstreams task
    changeStreet(&addr);
```

```
    swapEntries(&bigArray);
}
```

Resulting AutoStreams firmware, using the argument `--stage-count 4` :

```
void main() {
    changeStreet(&addr);
#pragma microstreams task
    swapEntries(&bigArray);
#pragma microstreams task
    changeStreet(&addr);
    changeStreet(&addr);
#pragma microstreams task
    swapEntries(&bigArray);
}
```